ENCICLOPEDIA PRACTICA DE LA

INFORMATICA APLICADA

SINCLAIR AMSTRAD COMMODORE IBM

3

Programación estructurada en el lenguaje Pascal

Juan Ignacio Puyol



EDICIONES SIGLO CULTURAL

ENCICLOPEDIA PRACTICA DE LA

INFORMATICA APLICADA

3

Programación estructurada en el lenguaje Pascal

EDICIONES SIGLO CULTURAL

Una publicación de

EDICIONES SIGLO CULTURAL, S.A.

Director-editor:
RICARDO ESPAÑOL CRESPO.

Gerente:
ANTONIO G. CUERPO.

Directora de producción:
MARIA LUISA SUAREZ PEREZ.

Directores de la colección:
MANUEL ALFONSECA, Doctor Ingeniero de Telecomunicación y Licenciado en Informática
JOSE ARTECHE, Ingeniero de Telecomunicación

Diseño y maquetación:
BRAVO-LOFISH.

Dibujos:
JOSE OCHOA Y ANTONIO PERERA.

Tomo 3. **Programación estructurada en el lenguaje Pascal**
JUAN IGNACIO PUYOL, Ingeniero Industrial

Ediciones Siglo Cultural, S.A.

Dirección, redacción y administración:
Sor Angela de la Cruz, 24-7.º G. Teléf. 279 40 36. 28020 Madrid.

Publicidad:
Gofar Publicidad, S.A. Benito de Castro, 12 bis. 28020 Madrid.

Distribución en España:
COEDIS, S.A. Valencia, 245. Teléf. 215 70 97. 08007 Barcelona.
Delegación en Madrid: Serrano, 165. Teléf. 411 11 48.

Distribución en Ecuador: Muñoz Hnos.

Distribución en Perú: DISELPESA.

Distribución en Chile: Alfa Ltda.

Importador exclusivo Cono Sur:
CADE, S.R.L. Pasaje Sud América. 1532. Teléf.: 21 24 64.
Buenos Aires - 1.290. Argentina.

ISBN del tomo: 84-7688-026-X.
ISBN de la obra: 84-7688-018-9.

Fotocomposición:
ARTECOMP, S.A. Albarracín, 50. 28037 Madrid.

Imprime:
TRAIMSA. Nicolás Morales, 38-40. 28019 Madrid.

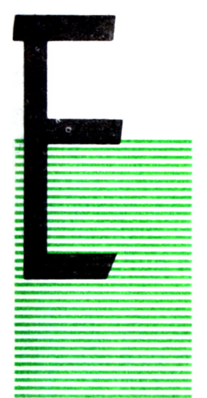
Depósito legal: M-34923-1986
Printed in Spain - Impreso en España.

Suscripciones y números atrasados:
Ediciones Siglo Cultural, S.A.
Sor Angela de la Cruz, 24-7.° G. Teléf. 279 40 36. 28020 Madrid

Octubre, 1986.
P.V.P. Canarias: 365,-

INDICE

INTRODUCCION 1



EN contra de lo que la mayoría de la gente supone, los ordenadores son máquinas sin inteligencia alguna. De lo único que son capaces es de seguir instrucciones que se les han dado previamente en forma de programa para realizar ciertas tareas. Por poner un ejemplo, una lavadora del modelo más sencillo es también una máquina con unos programas que le permiten realizar diferentes tareas según sean los requisitos de la colada. En este aspecto la única diferencia entre ambos tipos de máquina es que los programas de ordenador pueden tener miles de instrucciones y éstas se ejecutan a un ritmo de miles, o millones, de veces por segundo.

Si en algún momento un ordenador puede parecer inteligente, es porque previamente se le ha dotado de un programa que le indica qué acciones emprender en las diferentes situaciones que se puedan producir. Si hubiera que dar a un ordenador algún calificativo propio de un ser vivo, éste sería el de obediente, por lo que si se junta esto a la capacidad de ejecutar programas con muchas instrucciones a un ritmo muy rápido, pueden llegar a realizar tareas inabordables por un ser humano.

Si el programa no cubriera todos los casos que se pudiesen presentar, podría llegar un momento en que el ordenador no supiese qué hacer a continuación, aunque pudiera parecer evidente. Si las instrucciones fuesen erróneas, por obvio que esto resultara, el ordenador las ejecutaría. La máquina no sabe lo que está haciendo; simplemente se limita a seguir las instrucciones que se le han dado.

Cuando un programa es complejo es muy difícil tener la certeza absoluta de que está libre de errores y de que cubre todas las posibles situaciones que se puedan presentar. Lo mejor que se puede hacer, por tanto, es escribir los programas de manera disciplinada y utilizando técnicas especialmente pensadas para hacer los programas claros y fáciles de revisar y corregir. Este es el objetivo de la programación estructurada.

Las instrucciones que entienden los ordenadores se expresan por medio de números. Es lo que se conoce como *código máquina* y, en los primeros tiempos de la informática, los programas se escribían utilizando los números de las diferentes instrucciones directamente. Posteriormente se empezó a utilizar lo que se denomina *lenguaje ensamblador,* que consiste en usar, en lugar de cada número de instrucción, un nombre simbólico que recuerde para qué sirve. Por ejemplo, si la instrucción con código 17 54 significa «ir al punto 54 del programa», se podría indicar como *«saltar a 54»* en lugar de utilizar sólo números. Posteriormente, un programa de ordenador denominado *programa ensamblador* se encargaría de traducir una a una las instrucciones de ensamblador a instrucciones de código máquina para así generar el programa definitivo.

Las instrucciones de código máquina permiten, por separado, hacer sólo tareas muy sencillas, del estilo de «sumar este número a este otro», «si tal número es menor que 5 entonces repetir las tres instrucciones anteriores», etc.

Si quisiésemos entonces expresar, dentro de un programa, tareas más complicadas, como presentar un número en la pantalla del ordenador o calcular el seno de un ángulo, para cada una de ellas habría que preparar un montón de instrucciones de código máquina.

Para simplificar esto se crearon lo que se denomina *lenguajes de nivel alto,* uno de los cuales es el PASCAL. En programas escritos con ellos, tareas como las anteriores se expresarían como *«presentar (34)»* o *«seno (alfa)»,* utilizándose, en general, instrucciones más parecidas a las que se podrían dar a un ser humano.

Luego, la ejecución de cada una de estas instrucciones complejas supondría la ejecución de todas las de código máquina que fueran necesarias para realizar su cometido.

Hay básicamente dos formas de conseguir esto, por medio de lo que se denomina «ejecución bajo intérprete», o por medio de un proceso previo de «compilación».



INTERPRETES Y COMPILADORES

Supongamos que el ordenador fuera un operario extranjero al que hubiera que darle las instrucciones, para hacer su trabajo, escritas en su idioma, el croata.

Como nosotros no sabemos croata, las instrucciones las tenemos escritas en castellano. Por ejemplo:

1. Fijar la pieza en el tornillo de mesa.
2. Coger un tornillo del cajón A.

3. Ponerlo en el agujero número 3 con una arandela de presión.
4. Apretar con una llave de tubo del 10.
5. ...

Para conseguir que el operario las ejecute, tenemos dos alternativas:

1. Contratar a un intérprete para que vaya leyendo una instrucción tras otra y repitiéndoselas al operario en croata, a medida que las vaya ejecutando.
2. Contratar a un traductor para que prepare un libro con todas las instrucciones traducidas al croata. Una vez hecho esto, el libro será entregado al operario para que empiece a ejecutarlas sólo.

Las ventajas de la solución con intérprete serían:

— Con el intérprete el operario se puede poner a trabajar nada más se haya terminado de preparar o modificar las instrucciones en castellano. Con el traductor habría que esperar primero a que se tradujeran todas las instrucciones y a que se imprimiese el libro, y eso cada vez que se introdujera un cambio.
— Con el intérprete, es posible hacer que el operario deje un momento de trabajar, e interrogarle para saber cómo se encuentra y si las instrucciones son correctas y lo suficientemente claras, tras lo cual seguiría trabajando. Con el traductor, como tras hacer su trabajo se fue, no hay forma de entenderse con el operador mientras trabaja.

Por otra parte, las ventajas de la solución con traductor serían:

— El operador, al tener todo traducido, puede trabajar muy deprisa, sin tener que esperar a que un intérprete vaya traduciendo cada paso antes de ejecutarlo.
— La sala de trabajo puede ser más pequeña, pues sólo hace falta que esté el operador en ella.
— Como el traductor se puede tomar más tiempo para preparar el libro, las instrucciones traducidas por él son mucho más claras y concisas.

Este ejemplo ilustra de manera clara la diferencia entre las dos maneras de proceder.

Cuando se ejecuta un programa bajo intérprete, lo que está funcionando en el ordenador es un programa *intérprete* que va tomando una a una las instrucciones del nuestro y haciendo que el ordenador ejecute el paquete de instrucciones de código máquina que le corresponden.

Por el contrario, la otra alternativa consiste en, por medio de un programa *compilador* o traductor, generar uno nuevo a base de instrucciones de código máquina. Posteriormente será este nuevo programa el que pase a ejecutarse en el ordenador.

Por ello, el proceso de elaboración de un programa que va a ser compilado es, normalmente, mucho más laborioso. En primer lugar, hay que

escribir el programa en lenguaje de nivel alto, utilizando para ello lo que se denomina un programa *editor,* que permite redactarlo de manera cómoda. Tras ello, utilizando el programa compilador, habría que traducirlo a instrucciones de código máquina.

Por último, el nuevo programa así generado sería entregado al ordenador para que lo ejecutara. Si tras esto se comprobase que es necesario introducir cambios, para ello habría que volver a utilizar el programa editor, tras lo cual se volvería a compilar el programa, etc.

Sin embargo, los programas intérpretes suelen incluir un editor, por lo que, tras escribir nuestro programa, se puede pasar a ejecutarlo en cuestión de fracciones de segundo con sólo pulsar unas pocas teclas. Además, la ejecución puede ser interrumpida en cualquier momento para obtener información sobre su situación, y continuar después o pasar a introducir cambios, de manera prácticamente instantánea.

Debido a la facilidad de desarrollo de los programas utilizando un intérprete para su ejecución, y a la mayor velocidad y eficacia cuando se acude a la compilación, a veces existe para un mismo lenguaje la posibilidad de utilizar los dos sistemas. Así, se ejecutaría el programa con un intérprete durante el proceso de elaboración y una vez que éste se diera por terminado, se compilaría el programa y se pasaría a utilizar su traducción.

Aunque con el PASCAL al principio se utilizaba una mezcla de los dos sistemas (se traducía a un lenguaje intermedio que luego era ejecutado bajo intérprete), la mayoría de las versiones actuales son compiladas.

No obstante, y en tiempos recientes, han aparecido en el mercado programas compiladores de PASCAL que incluyen un editor y que permiten desarrollar y probar los programas prácticamente con la misma rapidez y comodidad que con un intérprete, teniéndose además todas las ventajas de la compilación. Gracias a ellos el PASCAL ha pasado a ser uno de los lenguajes más utilizados en la actualidad.



LENGUAJES ESTRUCTURADOS

Los programas escritos en código máquina están formados siempre por una secuencia de instrucciones que se ejecutan una detrás de otra. Algunas de éstas sirven para alterar la marcha normal del programa y «saltar» a otra instrucción distinta de la que hay a continuación, pudiéndose gracias a ellas repetir bloques de instrucciones y, si la alteración estuviera condicionada al resultado de algún cálculo previo, escoger diferentes instrucciones para ejecutar a continuación según sea ese resultado.

Los programas escritos con los primeros lenguajes de alto nivel que se desarrollaron y parte de los que se utilizan en la actualidad tienen, básica-

mente, el mismo aspecto que los programas escritos en código máquina, sólo que con instrucciones que suponen la realización de tareas más complicadas.

Por ejemplo, un programa para obtener y mostrar raíces cuadradas de números podría ser algo así:

1. Presentar un mensaje en la pantalla del ordenador pidiendo un número.
2. Leer el número tecleado.
3. Si el número es negativo, ir al punto 6.
4. Calcular la raíz cuadrada del número y presentarla.
5. Ir al punto 7.
6. Sacar mensaje recordando que el número debe ser positivo.
7. Sacar mensaje preguntando si se desea seguir.
8. Leer respuesta.
9. En caso de respuesta afirmativa, ir al punto 1.
10. Sacar mensaje de despedida.

Es posible demostrar que todo programa, por complejo que sea, puede expresarse combinando tres estructuras elementales:

— Secuencias de instrucciones que se ejecutan una detrás de otra.

— Repetición de grupos de instrucciones según una condición dada.

— Selección de unas instrucciones u otras según sea el resultado de un cálculo previo.

En el ejemplo se puede observar que, gracias a instrucciones del tipo «ir al punto tal», se dan las tres estructuras:

— Cuando la última respuesta es afirmativa se vuelve a repetir todo desde el principio.

— Según que el número sea positivo o no, se ejecutan unas instrucciones u otras.

— Las instrucciones se ejecutan una detrás de otra mientras no se indique lo contrario.

Sin embargo, su presencia no resulta evidente a primera vista y, si el programa completo constara de cientos o miles de instrucciones, podría resultar francamente difícil descubrir su arquitectura general cuando hubiera que revisarlo o modificarlo.

Los programas serían mucho más claros si, al escribirlos, se indicaran explícitamente las diferentes estructuras a utilizar. Por ejemplo:

Repetir la secuencia siguiente...

— Presentar un mensaje en la pantalla del ordenador pidiendo un número.

— Leer el número tecleado.

— Si el número es negativo...

 — sacar mensaje recordando que el número debe ser positivo.

 ... pero en caso contrario:

 — Calcular la raíz cuadrada del número y presentarla.

— Sacar mensaje preguntando si se desea seguir.

— Leer respuesta.

... hasta que la respuesta a la última pregunta sea negativa.
Sacar mensaje de despedida.

Los primeros lenguajes de programación de nivel alto, como el FORTRAN, dependían casi en exclusiva de las instrucciones de tipo «ir al punto tal» para obtener la repetición o selección de instrucciones, al igual que sucede con lenguajes más recientes como el BASIC. Con ellos, los programas se construyen prácticamente igual que programando en código máquina.

A finales de los años sesenta, y cobrando auge hasta nuestros días, empezó a surgir una corriente entre los teóricos de la informática, contraria a esta forma de construir programas, y partidaria del empleo de lenguajes «estructurados» que permitieran expresar las diferentes estructuras de un programa de manera clara, para así facilitar su comprensión y permitir su revisión y corrección de una manera más rápida y fiable. Uno de los primeros lenguajes de este tipo fue el ALGOL.

Para intentar acomodarse a las nuevas tendencias, pronto empezaron a surgir variantes de los lenguajes del primer tipo que permiten expresar las estructuras de tipo repetición y selección directamente, sin el empleo de instrucciones para saltar de un punto a otro del programa. A estas variantes se les ha llegado a poner nombres como «BASIC estructurado».

Sin embargo, el concepto de lenguaje estructurado va más allá de la posibilidad de prescindir de las instrucciones de tipo «salto».

Otra de las características más importantes de la programación estructurada es la posibilidad de escribir los programas, no como un chorro interminable de instrucciones, sino empezando por una descripción aproximada de las diferentes tareas a realizar y pasando luego a describir cada una de éstas, por separado, de manera más detallada. Por ejemplo, un pro-

grama para ordenar alfabéticamente los clientes de una empresa sería algo así:

— Pedir los nombres de los clientes.
— Ordenarlos.
— Presentarlos, ya ordenados, por la pantalla del ordenador.

Posteriormente, y por separado, se detallaría con mayor profundidad cada uno de esos puntos. Por ejemplo:

«Pedir los nombres de los clientes:»

Repetir la secuencia siguiente...

— Sacar un mensaje pidiendo el nombre de un cliente.
— Leer el nombre tecleado.
— Guardarlo en la memoria del ordenador.
— Sacar mensaje preguntando si se desea seguir.
— Leer respuesta.

...hasta que la última respuesta sea negativa.

Aquí se podría haber utilizado también una descripción superficial de alguno de los puntos, que sería a su vez detallado por separado.

El desglose de los programas en partes de menor complejidad, y éstas a su vez en otras, etc., permite, a la hora de revisar o modificar un programa, localizar rápidamente la zona de interés e introducir los cambios, teniéndose la seguridad de que otras partes del programa no se verán afectadas. Además, de esta manera es posible dividir cómodamente el trabajo entre diferentes personas o utilizar partes que ya se hubieran escrito con anterioridad para otro programa.

Aunque los lenguajes como el BASIC permiten, en cierta forma, esta descomposición de los programas por medio de lo que se denomina «subrutinas», la flexibilidad e independencia de las diferentes partes es mucho menor que cuando se emplea un lenguaje auténticamente estructurado.

En los lenguajes tipo BASIC, sólo se puede trabajar, básicamente, con números y caracteres (letras, cifras ...); en otras palabras, con prácticamente el mismo tipo de datos que pueden tratar los programas en código máquina.

Por ejemplo, para utilizar en un programa los meses del año, o los días de la semana, habría que asociar a cada uno de éstos un número, que sería el que realmente se utilizaría.

Por otra parte, en el caso de los clientes del ejemplo anterior, si hubiera que utilizar diferentes datos sobre ellos (nombre, saldo de la cuenta...), cada uno de estos datos habría que tratarlo por separado.

Otra de las características de los lenguajes estructurados es la posibilidad de trabajar con datos que reflejen de manera precisa qué es lo que, en

el mundo real, representan. En otras palabras, permiten prescindir del hecho de que los programas en código máquina sólo trabajan con números y caracteres. Por tanto, sería posible llamar a los diferentes meses o días de la semana por su propio nombre, o, de manera similar a como se hace al reflejar los diferentes datos de un cliente en una ficha de oficina, utilizar todos los datos de cada cliente como un conjunto indisoluble.

Esta última característica, denominada «estructuración de datos» y de la que veremos abundantes ejemplos a lo largo del libro, junto con las dos mencionadas anteriormente son, grosso modo, lo que diferencia a los lenguajes estructurados de los demás.



EL LENGUAJE PASCAL

El PASCAL, quizá el más representativo de los lenguajes estructurados, fue ideado hacia principios de los años setenta por el profesor Niklaus Wirth, del Instituto Federal de Tecnología de Zurich (Suiza), uno de los principales impulsores de la programación estructurada desde sus primeros momentos.

Cuando lo concibió, lo hizo con propósitos estrictamente académicos, tanto para ilustrar los conceptos de la programación estructurada, como para disponer de una herramienta adecuada para la enseñanza de la informática. Su descripción del lenguaje es lo que se denomina PASCAL «estándar».

Posteriormente, el PASCAL, de manera no prevista por su autor, comenzó a utilizarse en aplicaciones profesionales, por lo que fueron apareciendo nuevas versiones de programas compiladores de PASCAL que completaban el estándar para facilitar, fundamentalmente, la utilización de los diferentes dispositivos que se conectan a los ordenadores o las peculiaridades de un ordenador concreto.

Ha sido recientemente, con el auge de los ordenadores personales, cuando realmente el PASCAL ha pasado a establecerse como uno de los lenguajes más utilizados.

Posteriormente, se han creado nuevos lenguajes estructurados como el ADA, el C o el MODULA-2 (este último ideado también por el profesor Wirth), pero ninguno de ellos ha alcanzado todavía la difusión del PASCAL (aunque, en nuestra opinión, el MODULA-2 debiera empezar ya a ser su sucesor).



SOBRE EL LIBRO

Ya se ha mencionado que en el desarrollo de un programa PASCAL existen varias fases.

En primer lugar hay que redactar el programa. Este proceso es exactamente igual al de, por ejemplo, redactar una carta por medio del ordenador. En el mercado existe una gran variedad de programas editores para realizar esta tarea con la mayoría de los ordenadores existentes, y el lector debe acudir al manual de su editor concreto para aprender a utilizarlo.

A continuación, hay que realizar el proceso de compilación. Para ello, hay que poner en marcha en el ordenador el programa compilador de PASCAL que tengamos, y suministrarle el texto del programa que previamente hemos preparado.

A veces, dependiendo de cada compilador, hay que realizar tras la compilación lo que, en jerga informática, se denomina «linkedición».

Por último, se pasaría a ejecutar el programa en código máquina. Cada compilador se utiliza de una manera distinta, y en sus manuales siempre hay cantidad de ejemplos explicando claramente los pasos a realizar desde que se tiene el programa PASCAL escrito, hasta que se pone en marcha, por fin, su traducción.

A la hora de escribir este libro, se ha partido de la base de que el lector tiene, por lo menos, una mínima experiencia con ordenadores personales, y por ello no se ha dedicado ni una línea a explicar la utilización de un editor o compilador en concreto.

Sin embargo, se ha huido en todo momento de acudir a explicaciones que precisen un conocimiento más que superficial del funcionamiento de los ordenadores. Por ello, los lectores experimentados quizá encontrarán excesivamente simples las explicaciones de algunas cuestiones como, por ejemplo, el funcionamiento de los procedimientos recursivos, pues conceptos como el de «pila» no se mencionan en ningún momento.

Sólo se presupone que el lector está familiarizado con cuestiones básicas como, por ejemplo, que la pantalla del ordenador está organizada en filas y columnas, o que el espacio en blanco es un carácter más tan respetable como una letra o una cifra, o que la memoria del ordenador es una especie de colección de casilleros en los que se pueden guardar datos, tantos más cuanto mayor sea el número de aquéllos, etc.

En general, se ha procurado poner ejemplos escritos en PASCAL más o menos «estándar» para que sean traducibles por cualquier compilador con ninguno o pocos cambios; sólo cuando es inevitable se ha acudido a utilizar un compilador concreto, como, por ejemplo, al hablar de ficheros. Los ejemplos se han escogido, no pensando en desarrollar programas útiles (aunque algunos puedan serlo), sino más bien en ilustrar los diferentes conceptos que se van desarrollando. De paso, se ha aprovechado para introducir conceptos elementales como son la ordenación de tablas y el recorrido de estructuras tipo cola o árbol.

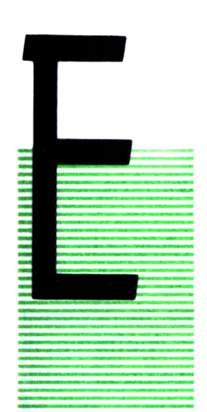
Por todo esto, los ejemplos no son siempre soluciones tan brillantes como las que, es de esperar, se le ocurrirán al lector cuando tenga un poco de práctica.

Por otra parte, se han marginado deliberadamente aspectos que, a priori, no son estrictamente necesarios para aprender a programar en PASCAL, como es el de variables empaquetadas («packed» en inglés).

Por supuesto, las posibles peculiaridades de un compilador, como pueden ser las instrucciones PEEK, POKE o INLINE para manejar las interioridades del ordenador, ni se mencionan.

El objetivo principal del libro es despertar una inquietud en lectores que ya hayan tenido alguna experiencia, pequeña o grande, con otros lenguajes de programación. Si esto se consigue, el lector deberá acudir a libros más especializados y rigurosos para profundizar en el PASCAL y la programación estructurada.

EL PROGRAMA PASCAL 2



STE capítulo hace una introducción al programa PASCAL, e ilustra cómo se definen los datos, cómo se leen datos del teclado y se escriben en la pantalla, y cómo se realizan cálculos. Antes de entrar en detalles veamos un ejemplo de programa con el mínimo de elementos posible:

```
PROGRAM PROGRAMATORPE;
BEGIN
 WRITE ('DOS Y DOS SON CINCO')
END.
```

Este programa escribe simplemente una frase (DOS Y DOS SON CINCO) en la pantalla del ordenador.

Palabras reservadas

En el ejemplo se observa que hay unas palabras impuestas por el PASCAL: PROGRAM, BEGIN y END. A éstas y a otras que irán apareciendo a lo largo del libro se les denomina *palabras reservadas,* es decir, palabras que sólo se pueden utilizar para cometidos específicos y que, por tanto, no se pueden utilizar nunca para dar nombre a variables, partes del programa, etc.

Identificadores

Hay también una palabra puesta a nuestro gusto: PROGRAMATORPE. Es lo que se denomina un *identificador.*

Los identificadores se utilizan en PASCAL para dar nombre a variables, programas, tipos de variables y, en general, a todos los elementos definidos por el programador.

A la hora de escribir un identificador, las únicas restricciones que se tienen son:

— Deben empezar por una letra.

— Tras esa letra puede ir cualquier combinación de letras y cifras. (Normalmente no se admite la Ñ ni los acentos.)

— No se admiten espacios en blanco de por medio.

— En general, aunque esto depende del compilador, se admite cualquier longitud, pero para diferenciar dos identificadores sólo se suelen tener en cuenta los primeros ocho caracteres. Así, los identificadores MURCIELAGO y MURCIELA son iguales para el compilador.

— Normalmente se admite la mezcla de letras mayúsculas y minúsculas.

Según estas reglas, H, PEPE y T37H son identificadores válidos, mientras que 3TJ, PE/PE y 7 son incorrectos.

Algunos compiladores admiten también el carácter de subrayado, por lo que FICHA_UNO sería también un identificador válido.

WRITE es un *identificador predefinido,* es decir, algo que ya tiene un significado sin que nosotros se lo hayamos dado previamente. A diferencia de las palabras reservadas, los identificadores predefinidos sí se pueden utilizar para algo nuestro, perdiendo entonces su significado previo, que es en este caso hacer que se escriba algo por la pantalla.

Para no complicar las cosas, desde ahora se tratarán los identificadores predefinidos como si fueran palabras reservadas, o sea, se utilizarán para su cometido original.



ESTRUCTURA DEL PASCAL

El propósito de un programa es, en general, procesar un conjunto de datos de entrada para obtener unos datos de salida. Un programa PASCAL consta así de dos partes, una en la que se describen los datos a utilizar y otra en la que están las instrucciones para procesar esos datos.

El programa siempre empieza por una cabecera que consta de la palabra reservada PROGRAM seguida del nombre del programa, que puede ser cualquier identificador válido. La cabecera se separa de lo que venga a continuación mediante un punto y coma.

Tras la cabecera viene la descripción de los datos y tras ella la palabra reservada BEGIN (comenzar), que se utiliza para indicar el comienzo de la parte de instrucciones. Por último, END indica el fin de esa parte, y va acompañado de un punto para señalar que ahí acaba todo.

En definitiva quedaría así:

PROGRAM EJEMPLO;

Descripción de datos

BEGIN

Instrucciones

END.

En el ejemplo del principio del capítulo, debido a que no hay datos que procesar, no existe descripción de datos.

Igualmente podríamos tener un programa sin instrucciones, aunque, por supuesto, no serviría para nada:

```
PROGRAM INUTIL;
BEGIN
END.
```

El número de líneas utilizado para escribir un programa no significa nada en PASCAL. Se podría poner:

```
PROGRAM INUTIL; BEGIN  END.
```

o

```
PROGRAM INUTIL;
BEGIN  END.
```

y se tendría en todo momento el mismo programa.

De hecho, cualquier programa PASCAL podría escribirse en una sola línea lo bastante larga, aunque, ciertamente, sería difícil de descifrar luego y no es ese el objetivo del PASCAL.

Las palabras se deben escribir de una sola vez, sin partir con espacios por medio y sin dividirlas en dos líneas:

```
BE GIN
```

o

```
BE
GIN
```

El PASCAL interpretaría esto como dos palabras (BE y GIN). Igualmente, no se deben empalmar palabras:

```
BEGINEND
```

pues sería para el PASCAL una sola palabra.



DESCRIPCION DE DATOS

Veamos el siguiente programa:

```
PROGRAM UNO;       (* Primer programa con descripción de datos  *)

CONST
  ENE   = -14 ;    (* Vamos a obtener ENE al cuadrado y al cubo *)
  TEXTO = 'TRAS PENSARLO, HE OBTENIDO QUE' ;
VAR
  CUADRADO, CUBO : INTEGER ; (* Aquí se guardan los resultados *)

BEGIN
 CUADRADO := ENE * ENE;
 CUBO     := ENE * ENE * ENE;
 WRITELN (TEXTO);
 WRITELN ('EL CUADRADO DE ',ENE,' ES ',CUADRADO);
 WRITELN (TEXTO);
 WRITELN ('EL CUBO DE ',ENE,' ES ',CUBO)
END.
```

En este programa hay descritos dos tipos de elementos: constantes y variables. También tiene escritos unos comentarios.



Constantes

La declaración de constantes debe ser precedida por la palabra reservada CONST. Esto indica al compilador que lo que viene a continuación, hasta que se indique otra cosa, son definiciones de constantes.

Mediante ellas, asociamos un identificador a una constante o valor numérico o alfanumérico (una frase). Su único objeto es evitar tener que escribir el número o frase cada vez que se necesite y usar el identificador en su lugar. Si en algún momento se deseara cambiar el valor de una constante, bastaría con modificar su definición, mientras que si, por ejemplo, hubiéramos puesto -14 en lugar de ENE por todo el programa, habría que hacer muchos más cambios.

El valor de una constante no puede cambiarse durante la ejecución del programa.

En el ejemplo, a la palabra ENE se le ha asociado el valor -14; por ello, cada vez que aparezca ENE será como si se hubiera escrito -14 en su lugar. -14 es una constante del tipo denominado INTEGER o número entero.

Asimismo, a TEXTO le hemos asociado una frase o constante alfanumérica. Las frases se delimitan en PASCAL usando apóstrofes y pueden contener cualquier palabra, símbolo, espacios en blanco, etc., sin que el compilador intente analizarlas:

```
'Aquí pone BEGIN, END y PROGRAM sin que pase nada.'
```

Cuando la frase debe incluir algún apóstrofe, para evitar que éste marque el final se pone por duplicado:

```
'Esto es un '' que no marca final de frase'
```

Por supuesto, a la hora de presentar el texto en la pantalla aparecería un único apóstrofe.

No se deben confundir las constantes numéricas con las constantes alfanuméricas que tengan aspecto de número.

Las frases se guardan en la memoria del ordenador carácter a carácter usando unos códigos especiales para ello. Por ello, si declaramos P = '-14' esta constante alfanumérica se guarda en la memoria como un signo menos, un 1 y un 4, mientras que la constante ENE del ejemplo se guarda usando otros códigos diferentes que se emplean para guardar números.

Por ello, mientras ENE puede aparecer en cálculos matemáticos, P nunca podría. Es simplemente un texto.

Se pueden definir todas las constantes que se deseen, separando cada definición de la siguiente con un punto y coma. Tras cada identificador se debe escribir la constante asociada separada por un signo igual.

Es una buena práctica escribir una sola definición por línea.

Nótese que la palabra reservada CONST sólo se usa una vez. Todo lo que venga a continuación serán definiciones de constantes hasta que otra palabra reservada (VAR, BEGIN ...) cambie esa situación.



Variables

La definición de datos que pueden variar a lo largo de la ejecución del programa va precedida por la palabra reservada VAR.

Los datos variables se guardan en porciones de la memoria del ordenador, siendo estas porciones de mayor o menor tamaño según el tipo de dato que vayan a albergar.

La declaración de variables sirve para que:

1. El PASCAL reserve las porciones necesarias del tamaño adecuado.
2. Asignar a cada porción un nombre o identificador a fin de poderlas usar luego llamándolas por su nombre.

Es como si antes de utilizar los buzones de correo del portal de una casa hubiera que decir: «Hace falta un buzón de tal tamaño a nombre de los señores de Pérez, otro el doble de grande a nombre de la señora de García...».

Todas las variables en PASCAL deben ser descritas antes de utilizarlas. En PASCAL, tras la palabra VAR se pone el identificador de cada variable seguido de dos puntos y a continuación el tipo de variable.

Supongamos que para hacer un programa de contabilidad de una comunidad de vecinos necesitáramos guardar, al hacer los cálculos, el piso, el portal y la letra del vecino en cuestión.

Necesitaríamos dos porciones de memoria para guardar el piso y el portal, que serán números enteros, y otra para guardar la letra, es decir, un carácter:

```
VAR  PISO   : INTEGER;
     PORTAL : INTEGER;
     LETRA  : CHAR;
```

Para evitar escribir mucho, las variables de un mismo tipo se pueden agrupar así:

```
VAR   PISO, PORTAL  : INTEGER;
      LETRA : CHAR;
```

que es lo que se ha hecho en el programa UNO.

INTEGER y CHAR son dos tipos de variables predefinidos que sirven para guardar, respectivamente, números enteros y caracteres aislados (no frases).

En capítulos posteriores veremos cómo podemos definir nuestros propios tipos de variables.

A diferencia de otros lenguajes de programación, en PASCAL las variables no tienen valores conocidos cuando empieza a funcionar un programa; será durante su ejecución cuando se empiece a guardar datos en ellas.

Una de las características más importantes del PASCAL es que nunca se pueden mezclar datos de diferente tipo. Por ejemplo, antes de asignar un valor a una porción de memoria reservada para guardar números enteros, se comprueba que lo que se quiere guardar en ella es un número entero, avisando en caso contrario al programador con un mensaje de error. Esto impide que los programadores usen «trucos» que, aunque en un momento dado pueden ser útiles, sirven básicamente para hacer más farragosos los programas.

Otra característica muy útil es la capacidad de limitar los posibles valores que pueda tomar una variable. Así, si en el caso anterior sólo pudiera haber pisos del 1 al 15, podríamos poner:

```
VAR PISO : 1..15;
```

Esto hará que, si en algún momento se pretende guardar en la porción de memoria conocida como PISO un valor no comprendido entre el 1 y el 15, se avise con un mensaje de error. PISO queda definida así como variable de tipo subrango, es decir, sigue siendo entera, pero con valores limitados.

Igualmente se podría hacer con LETRA:

```
LETRA : 'A'..'E';
```

si sólo hubiera letras de la A a la E en la casa de que se trata.

ñar. Si es un número, hay que obtener los símbolos que lo representan previamente.

También para esto se dispone de procedimientos adecuados.



Salida

Se pueden enseñar datos por pantalla, utilizando los procedimientos WRITE y WRITELN (*write* significa escribir, en inglés).

La única diferencia entre ellos estriba en que, usando WRITE, lo que se escriba en la próxima ocasión aparecerá en la misma línea justo a continuación de lo escrito en este momento, mientras que con WRITELN nos aseguramos de que lo que se escriba después salga en la línea siguiente. Veamos un ejemplo:

```
PROGRAM DOS;
CONST
  ULTIMALETRA = 'E';
  ULTIMOPISO  = 15;
BEGIN
 WRITE   ('La última letra es la ');
 WRITELN (ULTIMALETRA);
 WRITELN;
 WRITE   ('y el último piso es el ');
 WRITELN (ULTIMOPISO)
END.
```

En la pantalla saldrá :

```
La última letra es la E

y el último piso es el 15
```

Lo que se desea que salga por pantalla se escribe entre paréntesis a continuación del nombre del procedimiento escogido, separando esta instrucción de la siguiente por medio de un punto y coma.

Se pueden escribir por pantalla:

— Constantes: WRITE ('PEPE') WRITE (17)
— Constantes declaradas previamente : WRITE (ULTIMOPISO)
— El valor de una variable en ese momento.

— Resultados de expresiones matemáticas y de otro tipo que ya se verán.

Si hay más de un dato para mostrar en la misma línea se puede utilizar una sola instrucción escribiendo los datos separados por comas:

```
WRITELN ('La última letra es la ',ULTIMALETRA);
```

Cuando se escriben números enteros, se ocupan tantos espacios de una línea como sea necesario. Así:

```
WRITELN (1,10,345);
```

produce como salida 110345. Esto se podría mejorar poniendo:

```
WRITELN (1,' ', 10,' ', 345 );
```

que daría 1 10 345.

Si quisiéramos que los números ocuparan una cantidad precisa de espacios, ésta se especificaría poniéndola a continuación del dato separada por dos puntos:

```
WRITELN (1:2 ,10:6 ,ULTIMOPISO:4);
```

daría 1 10 15.

En el programa DOS se usa también WRITELN sin datos. Esto es como escribir una línea vacía y sirve, por tanto, para dejar líneas en blanco.

En el programa UNO hay más ejemplos de utilización de WRITE y WRITELN.

Dependiendo del compilador de que se disponga existen otros procedimientos adicionales para la salida de datos:

— CLRSCR (contracción de Clear Screen, borra pantalla) sirve para limpiar la pantalla de cualquier cosa que pudiera haber escrita en ella. En otros compiladores este procedimiento puede adoptar un nombre distin-

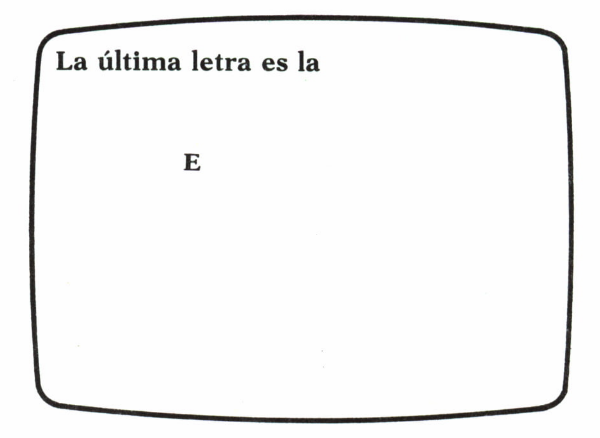
to, como, por ejemplo, PAGE. Es algo que se debe consultar en el Manual de nuestro compilador.

— GOTOXY (go to XY, ve a XY) utilizado por delante de WRITE o WRITELN sirve para que los datos aparezcan en un lugar escogido de la pantalla.

Usando estos dos procedimientos en el programa DOS tendríamos:

```
BEGIN
 CLRSCR;                    (* o PAGE, para borrar la pantalla *)
 WRITE ('La última letra es la ');
 GOTOXY (10,4);
 (* Lo siguiente saldrá en la columna 10 de la fila 4 *)
 WRITELN (ULTIMALETRA);
      -
      -
      -
```

Si el recuadro marcara los límites de la pantalla saldría:



Es decir, se debe poner el número de columna o valor de coordenada X y a continuación el número de fila o valor de coordenada Y separado por una coma. Estos valores se pueden indicar con constantes, como en el ejemplo, con variables o incluso con expresiones matemáticas.

Se ha supuesto que las columnas y filas se empiezan a numerar por 1, aunque en algunas versiones de PASCAL se empieza por 0.

Al igual que con CLRSCR, puede darse el caso de que este procedimiento tenga un nombre distinto o incluso que la columna y fila se deban poner en orden contrario.

Todas estas variaciones se pueden dar debido a que este procedimiento no está incluido en la definición estándar del PASCAL y, por tanto, está hecho a gusto de las personas que hayan desarrollado el compilador. En estos casos es necesario acudir al Manual para saber cómo utilizarlos, los valores de fila y columna que podemos usar, etc.



Entrada

La forma habitual de hacer que datos introducidos por el teclado se almacenen en una porción específica de memoria o variable es usar los procedimientos READ y READLN (*read* significa leer).

En general, la única diferencia entre ambos estriba en que, cuando se utiliza READLN, una vez se ha llenado la variable con el dato tecleado, todo lo que se hubiera escrito en la misma línea hasta que se pulsó la tecla de RETURN (o ENTER o INTRO o NEWLINE según el ordenador) se pierde y no se puede aprovechar en una lectura posterior. Utilizando READ, lo que resta en la línea queda disponible para las sucesivas instrucciones READ.

[Nota: utilizando el TURBO PASCAL con el ordenador personal IBM o compatibles, para que READ funcione de esta manera es necesario poner al principio del programa la opción de compilación, por ejemplo (*$G128*) para hacer la lectura a través del sistema operativo.]

Veamos un ejemplo:

```
PROGRAM TRES;
VAR    LETRA1, LETRA2 : CHAR;
       N: 1..1000;
BEGIN
 WRITELN ('Teclee dos letras (y pulse RETURN )');
 READ (LETRA1);
 READLN (LETRA2);
 WRITELN ('Ha pulsado la ',LETRA1,' y la ',LETRA2);
 WRITE  ('Teclee un número del 1 al 1000 ( y RETURN ) : ');
 READLN (N);
 WRITE ('N= ', N:5)
END.
```

Debido a la utilización de READ para LETRA1, la segunda letra puede teclearse en la misma línea, sin separarla con RETURN de la primera, pues seguirá disponible para su posterior lectura con READLN. Si se hubiera uti-

lizado READLN para LETRA1, la segunda letra se habría perdido y se tendría que volver a escribir (seguida de RETURN, que es la forma de indicarle al ordenador que ya se ha terminado de escribir el dato).

En el ejemplo también se hace la lectura de un número entero. Si el número tecleado fuera incorrecto (por ejemplo, 3A5) o estuviese fuera de rango (132700), el programa se pararía dando un aviso de error. En casi todos los compiladores de PASCAL es posible evitar la parada del programa y detectar el error para así dar la posibilidad de repetir la entrada de datos por medio de técnicas especiales que dependen de cada compilador y que escapan del alcance de este libro.

Si varios datos van a ser leídos de una sola vez, como sucede con LETRA1 y LETRA2, es posible hacerlo con una sola instrucción:

```
READLN (LETRA1,LETRA2);
```

El uso de READLN hará que todo lo que se haya tecleado más allá de las dos letras se ignore.

No obstante, la forma más adecuada de leer datos de teclado suele ser leerlos de uno en uno y usar WRITE o WRITELN previamente para indicar qué es lo que se pide en cada caso. En el programa TRES:

```
BEGIN
 WRITE ('Primera Letra: ');
 READLN (LETRA1);
 WRITELN ('Segunda: ');
 READLN (LETRA2);
 WRITELN ('Las letras son ',LETRA1,' y ',LETRA2);
  -
  -
  -
```

Nótese el diferente sitio en que aparecen las letras al ser tecleadas debido a la utilización de WRITE y WRITELN.



Expresiones

El programa anterior simplemente leía datos y los presentaba sin hacer ningún proceso con ellos ni generar nuevos datos. Evidentemente esto sirve para muy poco. Mediante el uso de *expresiones* se pueden generar nuevos valores a partir de datos previos.

Una expresión es una lista de variables y constantes combinadas mediante *operadores* para dar un resultado que a su vez puede ser guardado en una variable, mostrado en la pantalla o impresora, etc.

En una expresión **SOLO** se pueden combinar constantes y variables de un mismo tipo. No tiene sentido sumar un número a una letra.

En principio, hay cinco operadores que se pueden utilizar en expresiones de tipo INTEGER. Se escriben como + , –, * , DIV y MOD:

+ y – hacen que se sumen o resten los valores que se encuentran por delante y detrás de ellos: 3+1 equivale a poner 4.

La multiplicación se indica con * : 2 * 3 es igual a 6.

DIV representa la división entera, es decir, sin obtener decimales. 7 DIV 3 da el resultado 2, igual que 6 DIV 3.

MOD es el resto de la división entera. 7 MOD 3 vale 1, mientras que 6 MOD 3 da 0.

Las expresiones de tipo INTEGER se calculan empezando por la izquierda, pero realizando las sumas y restas después de las demás operaciones. Por ejemplo:

```
2 +  6 – 5    *   4 +   7 DIV 2        primero * y DIV :
2 +  6 –     20     +      3           luego + y – :
   8    –    20     +      3
       –12          +      3
             –9
```

Sin embargo, se puede alterar el orden de cálculo poniendo entre paréntesis lo que se desea que se calcule en primer lugar:

```
2 +  (6 - 5)  *  (4 + 7   DIV 2)       primero lo de los paréntesis:
2 +     1     *     7                  luego * y DIV :
2 +           7
   9
```

Se pueden utilizar todos los paréntesis que se quiera e incluso paréntesis dentro de paréntesis. Es una buena práctica poner paréntesis en cuanto haya la más mínima duda sobre el orden de cálculo de una expresión, pues, además, la expresión queda mucho más clara.

Una expresión de un tipo dado se puede poner en cualquier lugar en que pueda ir una variable o constante del mismo tipo. Por tanto, se podría escribir:

```
GOTOXY (1 + 1 , PISOULTIMO * PISO )
```

siendo PISOULTIMO y PISO, respectivamente, una constante y una variable de tipo INTEGER aunque, claro está, el resultado de PISOULTIMO * PISO debe dar un valor razonable.

Mediante la utilización de expresiones podemos cambiar el programa del principio del capítulo a:

```
PROGRAM PROGRAMATORPE;
BEGIN
 WRITE ('DOS Y DOS SON ', 2+2 )
END.
```

que evidentemente es menos torpe.

Nota: En expresiones de un tipo dado pueden aparecer términos de un subrango de ese mismo tipo.



Asignación

Las instrucciones de asignación permiten guardar datos en las variables.

Constan de dos partes separadas por el operador de asignación. A la izquierda se escribe el nombre o identificador de la variable en que se desea guardar el dato y a la derecha la constante, variable o expresión cuyo valor se desea guardar.

El operador se escribe como := , o sea, dos puntos inmediatamente seguidos de un signo igual.

El valor a guardar DEBE ser del mismo tipo que la variable en que se guarda. No se pueden mezclar nunca tipos distintos. Veamos un ejemplo:

```
PROGRAM CUATRO;
CONST   DOS = 2;
VAR     N: INTEGER;
        C: CHAR;
BEGIN
 (* Por ahora , N y C tienen valores desconocidos *)
 N:=  7 ;
 C:= 'A';
 (* Ahora ya hay valores conocidos en esas variables *)
 WRITELN ('N vale ',N,' y C vale ',C);
 (* Ahora el valor de N, que es 7, se suma a 1, y el resultado,
  8, se guarda en N que por tanto pasará a valer 8 *)
```

```
N:=N+1;
WRITELN ('N vale ahora ',N);

N:= N DIV DOS + 2;
(* La expresión vale 6, que será el nuevo valor de N *)
WRITELN ('N vale ahora ',N);

C:=C ; (* El valor de C se guarda en C: sólo sirve para
         perder el tiempo *)
END.
```

Está claro que para guardar un valor en una variable antes hay que obtenerlo. Por ello, en los casos en que la variable a la que vamos a asignar el valor aparece también a la derecha, se utiliza su valor anterior para obtener el resultado de la expresión, y sólo una vez calculada ésta pasará la variable a tener el nuevo.

Así, la instrucción N:=N+1 toma el valor anterior (7 en el ejemplo) para calcular la expresión. Una vez hecho esto, el resultado (8 en el ejemplo) se guarda en N.

Si a continuación hubiera otra instrucción N:=N+1, N pasaría a valer 9, etc.

Este caso concreto de instrucción resulta muy útil para contar el número de veces que se pasa por una parte de un programa. A las variables de tipo INTEGER así utilizadas se les denomina a veces *contadores de programa.*

En otros lenguajes como el BASIC esta instrucción se escribe como N=N+1, que tiene el aspecto de una expresión matemática que llevaría a la conclusión de que 1 es igual a 0.

Como en PASCAL se pretende eliminar cualquier ambigüedad, la asignación tiene su propio operador distinto del signo igual.

Para terminar, veamos un programa de ejemplo con todas las cosas estudiadas hasta ahora:

```
PROGRAM VECINOS;
CONST
  ULTIMOPISO = 15;
  ULTIMALETRA= 'E';
VAR
  PISO : 1..ULTIMOPISO;
  LETRA: 'A'..ULTIMALETRA;
  AGUA,PORTERO,ELECTRICIDAD,TOTAL,MES: INTEGER;
BEGIN
```

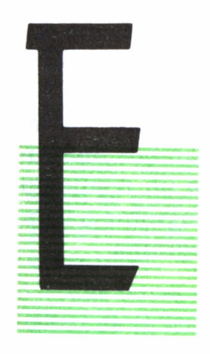
```
   CLRSCR;  (* Cambiar si nuestro PASCAL  no lo acepta *)
   WRITE ('Piso = '); READLN (PISO);
   WRITE ('Letra = '); READLN (LETRA);
   WRITE ('Gastos anuales de agua = '); READLN (AGUA);
   WRITE ('Gastos de Portería = '); READLN (PORTERO);
   WRITE ('Gastos de electricidad = '); READLN (ELECTRICIDAD);

   TOTAL := AGUA + PORTERO + ELECTRICIDAD ;
   MES   := TOTAL DIV 12;

   WRITELN (' La mensualidad del piso ',PISO,' letra ',LETRA,
            ' es ',MES);
   WRITELN ('Pero en Diciembre se acumulan los céntimos ');
   WRITELN ('de los meses anteriores : ',MES + TOTAL MOD 12)
END.
```

Nótese que la primera instrucción WRITELN ocupa más de una línea; esto se puede hacer siempre que no se parta un texto entre apóstrofes en dos líneas.

TIPOS PREDEFINIDOS DE DATOS 3



EN el capítulo anterior mencionamos los tipos de datos INTEGER y CHAR. Vamos a ver en este capítulo con más detalle los tipos existentes en PASCAL. En otros capítulos se verá cómo es posible además crear nuevos tipos adecuados a nuestras necesidades. Hay un capítulo dedicado especialmente al tipo REAL.



EL TIPO INTEGER

Este tipo es el que corresponde a los números enteros. Ya se ha visto cómo hacer para:

— escribir constantes:	100 , -15 , 0 , 1324
— declararlas:	CONST CIEN = 100
— definir variables:	VAR N : INTEGER
— escribir expresiones:	CIEN * (N - 15)

y además, cómo mostrar valores de tipo INTEGER, leerlos desde teclado y guardarlos en variables.

Los operadores permitidos son: + , – , * , DIV y MOD. Hay dos funciones importantes disponibles para este tipo de datos:

— Si escribimos ABS(X) , esto da como resultado el valor absoluto de X, o sea, X mismo si es positivo, o -X si es negativo.

— SQR (X) da como resultado el cuadrado de X (X*X). (SQR es la contracción de SQUARE, cuadrado en inglés).

Estas funciones se pueden utilizar como parte de expresiones sin restricción alguna. Por tanto, se podría escribir:

```
WRITELN ( 2 + ABS (3-7),'  y  ', SQR (2+1) ) ;
```

y esta instrucción mostraría en pantalla «6 y 9».

Los valores que se pueden manejar con constantes, variables y expresiones enteras tienen que estar comprendidos entre unos límites que dependen del ordenador y del compilador que tengamos. La constante predefinida MAXINT nos indica estos límites.

Por ello, si MAXINT (máximo entero) vale 32767, valor habitual, los valores de tipo INTEGER deben estar entre –32767 y +32767 ambos inclusive; por tanto, la expresión 1000 * 1000 , por ejemplo, daría un resultado que se sale de los límites provocando o bien la parada del programa o bien un resultado erróneo, según el compilador utilizado. El programa:

```
PROGRAM MAXIMOENTERO;
BEGIN   WRITELN ('El máximo entero posible es ',MAXINT) END.
```

nos indicaría el valor de MAXINT en nuestro PASCAL.

En los casos en que haya que tratar con cifras decimales o con números que se salgan de los límites, habrá que acudir al tipo REAL que se describe en capítulo aparte.



EL TIPO CHAR

Este tipo es el utilizado para manejar caracteres sueltos, es decir, una letra, una cifra, un signo de puntuación o incluso el espacio en blanco.

Las constantes de tipo CHAR se escriben poniendo el carácter en cuestión entre apóstrofes: 'Z'

Al igual que con el tipo INTEGER las constantes de tipo CHAR se pueden declarar al comienzo del programa:

```
CONST LETRADELPISO = 'E';
```

y también se pueden definir variables de tipo CHAR y mostrar y leer de teclado valores de este tipo (es decir, caracteres).

El conjunto de caracteres disponible depende del ordenador pero SIEMPRE existen, al menos, las mayúsculas del alfabeto inglés:

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.

las nueve cifras decimales:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

y el espacio en blanco.



Ordinales

Internamente, y sin que nosotros nos demos cuenta de ello, los caracteres se guardan en memoria utilizando números, uno distinto para cada posible valor. Son lo que se denomina *números ordinales* del conjunto de caracteres.

Las letras están ordenadas según estos números. Si, por ejemplo, el ordinal de la letra A fuera el 65, el de la B sería el 66 y así hasta la Z. Lo mismo sucede con las cifras del 0 al 9.

Si se quisieran utilizar estos números en expresiones de tipo entero, la función ORD (C), donde C es un valor cualquiera de tipo CHAR, proporciona un resultado de tipo INTEGER igual al ordinal del carácter.

A la inversa, si I fuera un valor de tipo INTEGER, CHR (I) nos proporcionaría el carácter cuyo ordinal es I. Veamos un ejemplo:

```
PROGRAM UNO;
VAR  I : INTEGER ;
     C : CHAR    ;
BEGIN
 C:= 'A';
 I:=ORD (C);
 WRITELN ('El ordinal de la letra ',C,' es ',I);
 WRITELN ('La letra cuyo ordinal es ',I+7,' es ',CHR(I+7));
 WRITELN ( 'El ordinal de la cifra 0 es ',ORD ('0'))
END.
```

Existen también dos funciones más :

— PRED (C) donde C es un carácter, nos devuelve el carácter anterior a él. PRED ('C') equivale a poner 'B'.

— SUCC (C) devuelve el siguiente a C.

Se puede comprobar que:

PRED (C) equivale a poner CHR (ORD(C) - 1) y
SUCC (C) a CHR (ORD(C) + 1).

Normalmente, en los ordenadores personales se utilizan los códigos ASCII (siglas de American Standard Code for Information Interchange, pronunciado «aski»), en que las letras empiezan por el ordinal 65, las cifras por el 48 y el espacio en blanco, por ejemplo, tiene el 32.



EL TIPO BOOLEAN

En capítulos posteriores veremos que, normalmente, los programas tienen que tomar decisiones en función de determinados resultados. Por ejemplo:

«Si el saldo bancario es mayor que 10000 ptas., entonces hacer esto...»

Es decir, hay que hacer la operación de comparar el saldo bancario con 10000 para ver si es mayor o no. Si fuera mayor, el resultado de la operación «saldo mayor que 10000» sería CIERTO y en caso contrario sería FALSO.

Las expresiones de este tipo, que pueden dar como resultado CIERTO o FALSO, se llaman expresiones LOGICAS o de tipo BOOLEAN.

En PASCAL está definido el tipo de dato BOOLEAN, que es aquél que sólo puede tener dos valores, CIERTO y FALSO, valores que se representan mediante las constantes predefinidas TRUE (cierto en inglés) y FALSE.

— Como tal tipo, podemos definir variables BOOLEAN donde guardar estos datos y que sólo pueden tomar los valores TRUE y FALSE:

```
VAR SALDOCORRECTO : BOOLEAN;
```

— Se pueden asignar valores a esas variables:

```
SALDOCORRECTO:= FALSE;
```

— Se pueden mostrar valores BOOLEAN:

```
WRITELN (SALDOCORRECTO);
```

Esto último haría que se escribiera la palabra TRUE O FALSE según el valor de la variable.

Sin embargo, NO se puede utilizar READ o READLN con variables BOOLEAN.



Expresiones con valores INTEGER que dan resultado BOOLEAN

Muy frecuentemente se desean comparar números enteros para tomar decisiones, como en el caso del saldo.

Si el valor del saldo estuviera guardado en la variable SALDO, la expresión

```
SALDO > 10000
```

daría TRUE o FALSE como resultado según que SALDO fuera mayor o no que 10000. Podríamos poner, pues:

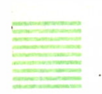
```
SALDOCORRECTO:= (SALDO > 10000);
```

Existen los siguientes operadores:

> significa «mayor que»:	SALDO > 10000
>= significa «mayor o igual que»:	(PISO-3) >= (2*4)
< significa «menor que»	
<= significa «igual o menor que»	
= significa «igual a»:	SALDO = 0
<>significa «distinto de»	ULTIMOPISO <> 15

Además, la función ODD (I) (odd significa impar entre otras cosas), donde I es un valor INTEGER, devuelve el valor TRUE si I es impar y FALSE en caso contrario:

```
ACERAIMPAR := ODD (PORTAL);
```



Expresiones con valores CHAR que dan resultado BOOLEAN

Dado que los caracteres tienen asociados unos números ordinales, las operaciones de comparación que se pueden hacer con el tipo INTEGER se pueden hacer también con el tipo CHAR, pues en el fondo lo que se comparan son los ordinales.

Así, 'A' < 'B' dará el resultado TRUE, pues ORD('A') es menor que ORD('B').

Podríamos escribir entonces:

```
ELPISOMAYOR:= (LETRA = 'A');
BARATO := (LETRA >= 'C');
```

Donde ELPISOMAYOR y BARATO son variables BOOLEAN. Por ejemplo, BARATO valdría TRUE si LETRA fuera C, D, E,...

En otras palabras, la comparación < , «menor que», se podría llamar ahora «viene antes, por orden alfabético, que» y de manera análoga para las otras.

Como se puede imaginar, esto será muy útil a la hora de ordenar alfabéticamente textos de cualquier tipo.



Expresiones de tipo BOOLEAN

Una expresión de tipo BOOLEAN es un conjunto de valores de ese mismo tipo combinados entre sí mediante operadores lógicos (o booleanos) para dar a su vez un resultado de tipo BOOLEAN. Los operadores disponibles son:

— AND, operación lógica «Y». La operación AND aplicada a dos valores lógicos da resultado TRUE si y sólo si ambos valores son TRUE simultáneamente.

— OR, operación lógica «O». Aplicada a dos valores lógicos resulta TRUE cuando cualquiera de ellos, o los dos a la vez, es TRUE.

— NOT, operación de negación. Aplicado a un valor lógico, proporciona su opuesto, es decir, FALSE si era TRUE y viceversa.

— En algunas versiones de PASCAL existe además la operación XOR u «O exclusivo», similar a OR, pero que resulta TRUE sólo si alguno de los dos valores es TRUE y no si ambos lo son a la vez. Esta operación, al no ser estándar, pudiera tener un nombre distinto.

Las expresiones se evalúan de izquierda a derecha, realizándose primero las operaciones NOT, luego las AND y después las demás.

Supongamos que las variables A, B y D tuvieran el valor FALSE y que C tuviera el valor TRUE. Entonces tendríamos:

A	OR	B	OR	C	AND	NOT (D)	primero NOT:
A	OR	B	OR	C	AND	TRUE	luego AND:
A	OR	B	OR		TRUE		y por último OR:
	FALSE		OR		TRUE		
			TRUE				

Al igual que en las expresiones de tipo INTEGER, se pueden utilizar paréntesis para asegurar un determinado orden de evaluación:

```
(A OR B) AND NOT (C OR D).
```

Volviendo al ejemplo del saldo anterior, si hubiera dos variables SALDO1 Y SALDO2 de tipo INTEGER que indicaran los saldos de dos cuentas de una persona, la instrucción:

```
HAYSUFICIENTE := (SALDO1 > 10000) OR (SALDO2 > 10000);
```

haría que la variable BOOLEAN HAYSUFICIENTE valiera TRUE cuando uno de los dos saldos al menos fuera mayor que 10000.

Nótese que la instrucción:

```
HAYSUFICIENTE := NOT ( (SALDO1 <= 10000) AND
(SALDO2 <= 10000) );
```

es equivalente: hay suficiente cuando NO sucede a la vez que SALDO1 es insuficiente y SALDO2 es insuficiente.

Nótese también que:

```
(SALDO1 > 10000)    equivale a    NOT (SALDO <= 10000) y
     (N = 100)      equivale a        NOT (N <> 100).
```

o sea, SALDO1 mayor que 10000 equivale a SALDO1 no menor o igual que 10000. Esto también sucede con los restantes operadores de comparación. Resumiendo en una tabla los resultados de las operaciones lógicas:

A	B	A AND B	A OR B	A XOR B	NOT A	NOT B
F	F	F	F	F	T	T
F	T	F	T	T	T	F
T	F	F	T	T	F	T
T	T	T	T	F	F	F

F = FALSE
T = TRUE

A y B pueden ser cualquier variable, comparación, etc., que proporcione un resultado de tipo BOOLEAN.

Notas:

— Para los amantes del BASIC: Como se ve, en PASCAL el resultado de una operación lógica sólo puede dar los valores TRUE y FALSE que NO son números. Por tanto, cosas como A = A + (B > 5), tan habituales en BASIC aprovechando que el resultado falso es un 0 y el cierto, generalmente, un -1, no son posibles en PASCAL.

NUNCA se pueden mezclar datos de diferente tipo.

— Aunque no resulta de demasiada utilidad, puede ser interesante saber que internamente los valores BOOLEAN, como los CHAR, tienen asociados números ordinales de manera que ORD (TRUE) es mayor que ORD (FALSE) y, por tanto, es posible comparar a su vez valores de tipo BOOLEAN. Ejemplos:

A = B, donde A y B son valores BOOLEAN, dará resultado TRUE si ambos son TRUE o FALSE a la vez.

A = FALSE dará TRUE si A es igual a FALSE, o sea, equivale a poner NOT (A). Igualmente A = TRUE o A <> FALSE equivale a poner A simplemente.

A <> B se puede comprobar que equivale a A XOR B mirando la tabla anterior.

Con la función ORD sí se podrían hacer cosas similares a las de BASIC, pues ORD da un resultado de tipo INTEGER:

```
A:= A + ORD (A <> 7)
```



EL TIPO STRING

Antes de nada: éste NO es un tipo estándar del PASCAL y, por tanto, su forma de utilización cambia de unas versiones a otras. Hay versiones de PASCAL que no tienen lo que aquí denominamos tipo STRING y que, por el contrario, llaman STRING a variables del tipo ARRAY OF CHAR, que sí son estándar y que veremos en otros capítulos.

Aquí nos limitaremos a describir sus características generales, debiéndose leer el Manual del compilador para las cuestiones de detalle.

Hemos visto en el capítulo anterior que se pueden definir constantes de tipo texto:

```
TEXTO = 'TRAS PENSARLO HE OBTENIDO QUE'
```

Sin embargo, las únicas variables vistas hasta el momento que permiten trabajar con caracteres, las de tipo CHAR, sólo permiten guardar caracteres sueltos.

En otros capítulos se verá que se pueden definir variables de tipo ARRAY o tabla que permiten manejar textos, pero éstos deben tener siempre la misma longitud, que se define al escribir el programa.

Para facilitar el manejo de frases es para lo que la mayoría de compiladores han incorporado el tipo STRING que, grosso modo, es similar al existente en BASIC.

La variable de tipo STRING permite guardar frases de longitud variable que puede ir desde cero a un límite que depende del compilador, normalmente 255 caracteres. En algunos compiladores se declaran así:

```
VAR NOMBRE : STRING;
```

mientras que en otros puede ser:

```
VAR NOMBRE : STRING [10];
```

caso de que 10 fuera el máximo número previsto de caracteres que pudieran llegar a tener los textos a almacenar en NOMBRE.

Algunos compiladores precisan formas más laboriosas de definición.

La asignación de datos a variables de tipo STRING se hace como siempre:

```
NOMBRE := 'PEPE' ;
NOMBRE := 'ANTONIO';
```

Un dato de tipo CHAR normalmente se puede guardar en un STRING:

```
NOMBRE := SUCC ('B');
```

Asimismo podemos asignar el STRING vacío:

```
NOMBRE := ''
```

Las variables de tipo STRING pueden ser leídas por medio de READ y READLN. También se pueden utilizar con WRITE y WRITELN:

```
PREGUNTA := 'Cuál es el nombre '; WRITELN (PREGUNTA);
READLN (NOMBRE);
```



Concatenación

La concatenación consiste en empalmar dos o más variables de tipo STRING. Se expresa utilizando el signo + . Así, la secuencia de programa:

```
NOMBRE := 'PEPE';
NOMBRE := NOMBRE + ' ' + 'LUIS ';
```

hace que NOMBRE contenga el texto 'PEPE LUIS'.

Normalmente, datos de tipo CHAR son utilizables en estas instruccio-

nes de concatenación. Sin embargo, sólo suele ser posible guardar un dato tipo STRING en una variable tipo CHAR si aquél tiene sólo un carácter.

Si en algún momento se pretende guardar en una variable un texto de longitud mayor que la admisible, sólo se guardará del texto la parte que quepa.



Comparación de strings

Igual que con el tipo CHAR se podían comparar dos caracteres, suele ser posible comparar dos strings.

Para ello, se comienza por comparar los primeros caracteres de ambos strings, y si fueran distintos el resultado de su comparación sería el que se tomaría como resultado final.

Si fueran iguales se pasaría al siguiente, etc., hasta llegar al final de alguno de los textos. En caso de igualdad de todos los caracteres comparados se supone «menor» al string más corto e iguales si la longitud coincide:

'ASNO' es menor que 'BURRO',
'ASNAS' es menor que 'ASNO' (hubo que llegar al cuarto carácter),
'ASNA' es menor que 'ASNAS' (decidido por longitud),
'ASNA' es igual que 'AS'+'N'+'A' y
'' (el string vacío) es menor que cualquier otro.

Como se ve, esto es directamente aplicable al proceso de ordenar alfabéticamente textos; así, ordenando de menor a mayor los anteriores resultaría: '', 'ASNA', 'ASNAS', 'ASNO', 'BURRO'. La comparación se expresa como en los casos vistos hasta ahora:

(NOMBRE > 'PEREZ') dará TRUE para los nombres que se encuentren, alfabéticamente, por detrás de PEREZ y FALSE en caso contrario.



Funciones para el tipo string

Casi todos los compiladores disponen de funciones para el manejo de STRINGS. Aquí vamos a describir un posible conjunto de funciones:

— LENGTH (NOMBRE) devuelve el número de caracteres (en tipo INTEGER):

LENGTH ('') vale 0 y
LENGTH ('ASNO' + 'S') vale cinco.

— POS (NOMBRE1,NOMBRE2) devuelve, en tipo INTEGER, la posición en que comienza el texto NOMBRE2 dentro del texto NOMBRE1, siendo 1 la posición del primer carácter:

POS ('BURRO','RR') vale 3 y

POS ('ASNO ','Z') vale 0 (no está).

— COPY (NOMBRE,P,N) devuelve la parte de NOMBRE que empieza en el carácter número P con N caracteres:

COPY ('PALABRA',3,4) vale 'LABR' y

COPY ('PALABRA',6,4) vale 'RA'

(al excederse la longitud del texto, se trunca).

Si, por ejemplo, la variable NOMBRE contuviera nombre y apellido separados por un espacio en blanco,

```
COPY (NOMBRE, 1, POS(NOMBRE,' ')-1)
```

devolvería el nombre y

```
COPY (NOMBRE, POS(NOMBRE,' ')+1,
      LENGTH(NOMBRE)- POS(NOMBRE,' '))
```

devolvería el apellido.

Supongamos que NOMBRE vale 'PEPE PEREZ'. POS (NOMBRE,' ') vale pues 5 y LENGTH (NOMBRE) 10, por lo que la primera expresión equivale a

COPY (NOMBRE,1,4), es decir, 'PEPE'

y la segunda a

COPY (NOMBRE,6,5), o sea, 'PEREZ'.

Con vistas a la claridad del programa, resultaría mejor guardar previamente POS (NOMBRE,' ') en una variable de tipo INTEGER.

```
P := POS (NOMBRE,' ');
```

y utilizar luego P en las otras instrucciones.

Estas funciones varían mucho de un compilador a otro, pero todos permiten hacer más o menos las mismas operaciones que se han descrito.

En cualquier caso, siempre podremos definir nuestras funciones, como se verá en otros capítulos.

Las funciones PRED, SUCC, Y ORD no son aplicables al tipo STRING.



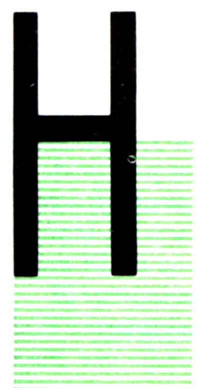
EL TIPO BYTE

Este tipo, existente sólo en algunos compiladores, es en la práctica un subrango 0..255 del tipo INTEGER.

Su interés radica en que el tamaño de la porción de memoria necesario es menor que en el tipo INTEGER.

Se cumple además que SUCC (255) es 0 y que PRED (0) es 255.

TOMA DE DECISIONES Y BUCLES 4



HASTA ahora, todos los programas que hemos visto han consistido en una secuencia de instrucciones simples que se ejecutaban siempre empezando por la primera, de una en una, hasta llegar a la última.

Si se desea construir programas más complejos, es necesario tener la posibilidad de escoger qué grupos de instrucciones se desea ejecutar en un momento dado y la de repetir bloques de instrucciones.

A continuación describiremos las más importantes estructuras de control existentes en PASCAL.



LA ESTRUCTURA IF

Esta estructura permite decidir durante la ejecución de un programa si una instrucción dada se debe ejecutar o no según una cierta condición. Opcionalmente, es posible escoger entre dos instrucciones. Veamos un ejemplo:

```
PROGRAM UNO;
 VAR N: INTEGER;
BEGIN
 WRITE ('Número cuyo cuadrado desea: ');  READLN (N);

 IF N<0 THEN WRITELN ('Ha notado que el número es negativo?');

 WRITELN (N,' al cuadrado = ', SQR (N));
 WRITELN ('Se acabó.')
END.
```

La instrucción WRITELN ('Ha notado...) sólo se ejecutará si N es menor que 0. Si traducimos del inglés:

IF	$N < 0$	THEN	WRITELN ('Ha notado...
"Si	N menor que 0	entonces	WRITELN ('Ha notado..."

se ve claramente cómo es la estructura.

Entre las palabras reservadas IF y THEN se escribe la condición, que debe dar un resultado de tipo BOOLEAN. Durante la ejecución del programa, al llegarse a la estructura IF se evalúa la expresión lógica y, caso de que el resultado sea TRUE, se pasa a ejecutar la instrucción cuya ejecución condicional se desea y que se encuentra tras la palabra THEN. Tras esto se sigue ejecutando lo que hubiera a continuación.

Si el resultado fuera FALSE, se pasaría a ejecutar lo siguiente directamente. En definitiva:

IF (condición) THEN (instrucción a ejecutar en caso de TRUE)

El conjunto así formado es una instrucción en sí misma, aunque al estar compuesta de varias partes se dice que es una instrucción «estructurada». Como tal instrucción, debe separarse de las que pudiera haber a continuación por un punto y coma.

Veamos otro ejemplo:

```
PROGRAM DOS;
 VAR N: INTEGER;
BEGIN
 WRITE ('Número cuyo cuadrado desea: ');  READLN (N);

 IF  ABS (N) > 100  THEN
    WRITELN ('No me gusta un número tan grande. Lo siento.')
  ELSE
    WRITELN (N,' al cuadrado = ', SQR (N));

 WRITELN ('Se acabó.')
END.
```

En esta otra variante de la estructura IF, sin embargo, se escoge entre dos posibles instrucciones según el resultado de la condición.

Tras la palabra reservada THEN se escribe aquélla que se desea ejecutar cuando el resultado de la condición ha sido TRUE. A continuación, y separada por la palabra reservada ELSE, se escribe la que hay que ejecutar en caso de ser FALSE el resultado.

Sea cual sea el resultado de la condición, y tras ejecutarse la instrucción correspondiente, se continúa con la siguiente a la estructura.

Una vez más, la traducción del inglés es clara:

IF	ABS(N) > 100	THEN	WRITELN ('No...
"Si	valor absoluto de N mayor que 100	entonces	WRITELN ('No...

ELSE	WRITELN (N,' al ...
en otro caso	WRITELN (N,' al ..."

Nótese que tras la primera instrucción y antes de ELSE no hay ningún punto y coma; si lo hubiera, al llegar a él el compilador, supondría que es el final de una estructura IF del primer tipo, con lo que al llegar a la palabra ELSE se produciría un error.

El conjunto formado por la condición, las dos instrucciones y las palabras reservadas IF, THEN y ELSE es, a su vez, una instrucción (estructurada, eso sí) y, por tanto, debe ir separada de la siguiente por un punto y coma.

La condición a evaluar puede ser cualquier expresión que proporcione un resultado de tipo BOOLEAN, es decir, TRUE o FALSE.

La instrucción (o instrucciones) cuya ejecución condicional se desea puede ser cualquiera que se nos ocurra, simple o estructurada. Entre éstas últimas se encuentra lo que se denomina *secuencia de instrucciones*.



Secuencia de instrucciones

Una *secuencia* o *bloque de instrucciones* es un conjunto de instrucciones de cualquier tipo escritas una detrás de otra, separadas entre sí por punto y coma, y enmarcadas por las palabras reservadas BEGIN, para indicar el comienzo, y END, para indicar el final. Por ejemplo:

```
BEGIN
 WRITELN;
 B:=2;
 IF A>2 THEN WRITE ('Pepe.');
 READ (C)
END
```

La secuencia en sí misma es una instrucción estructurada y como tal el **PASCAL** es muy claro respecto a su uso: se puede poner en cualquier sitio en que pudiera figurar una instrucción simple.

La ejecución de una secuencia consiste en ejecutar por orden las instrucciones que la integran.

Utilizando secuencias en una estructura IF es posible, por tanto, decidir si se ejecutan o no grupos complejos de instrucciones.
Veamos un ejemplo:

```
PROGRAM TRES ;
CONST
  ULTIMALETRA = 'E';
  ULTIMOPISO  = 15;
VAR
  LETRA : CHAR;
  PISO  : INTEGER;

BEGIN
 WRITELN ('Introduzca el piso: '); READLN (PISO);

 IF  (PISO <= 0)  OR  (PISO > ULTIMOPISO)   THEN
   BEGIN
     WRITELN ('Piso erróneo. Tomaré el primero.');
     PISO := 1
   END;           (* Aquí se acaba el conjunto IF...THEN... *)

 WRITELN ('Introduzca la letra del piso: '); READLN (LETRA);

 IF  (LETRA < 'A')  OR  (LETRA > ULTIMALETRA)   THEN
     BEGIN
       WRITELN ('Letra errónea.  Supongo la A.');
       LETRA := 'A'
     END
   ELSE
     WRITELN ('Bravo, es una letra correcta.');
     (* y aquí acabó el conjunto IF...THEN...ELSE... *)

 WRITELN ('El piso introducido es el ',PISO,'-',LETRA)
END.
```

(Nótese la indentación utilizada para hacer el programa más claro.)

Como la estructura IF en su conjunto es una instrucción estructurada, podría formar parte de una secuencia o ser incluso una de las instrucciones que forman parte de otra estructura IF. Por ejemplo, se podría escribir:

```
IF  (PISO > 0)  AND  (PISO <= ULTIMOPISO)  THEN

   IF  (LETRA >= 'A')  AND  (LETRA <= ULTIMALETRA) THEN
           WRITELN ('Piso y letra correctos.')
         ELSE
           WRITELN ('Sólo correcto el piso.');
```

Nota: En estos casos, el compilador, cuando encuentra la palabra reservada ELSE, supone que corresponde a la última estructura IF que no la tuviera todavía. Por ello, en este ejemplo lo que se tiene es una estructura IF-THEN-ELSE dentro de otra IF-THEN.

Si quisiéramos que ELSE correspondiera al primer IF (es decir, una estructura IF-THEN dentro de otra IF-THEN-ELSE), la solución sería:

```
IF  (PISO > 0)  AND  (PISO <= ULTIMOPISO) THEN
   BEGIN
    IF  (LETRA >= 'A')  AND  (LETRA <= ULTIMALETRA) THEN
            WRITELN ('Piso y letra correctos.')
   END
 ELSE
   WRITELN ('Piso erróneo. Ya, ni miro la letra.');
```

Utilizando una secuencia de una sola instrucción (lo que en principio podría parecer absurdo), las palabras BEGIN y END han hecho de «paréntesis» para así forzar a que ELSE corresponda a la primera estructura IF.



LA ESTRUCTURA REPEAT

Hay ocasiones en que se necesita repetir la ejecución de una cierta instrucción hasta que se cumpla una condición dada. Para ello se dispone de la estructura REPEAT. Empecemos con un ejemplo:

```
PROGRAM CUATRO;
 VAR  N: INTEGER;
BEGIN
 N:= 0;
```

```
REPEAT
  N:=N+1
UNTIL  N > 100;
WRITELN ('Se acabó.')
END.
```

La estructura REPEAT se compone de las palabras reservadas REPEAT y UNTIL, la instrucción a repetir escrita entre ellas, y, en último lugar, la condición por la que se debe terminar con las repeticiones y pasar a lo siguiente.

Durante la ejecución del programa, al llegarse a la estructura REPEAT, se ejecuta la instrucción que alberga, tras lo cual se pasa a evaluar la condición que hay tras la palabra reservada UNTIL. Si el resultado es TRUE, se pasa a lo que hubiera a continuación, pero si fuera FALSE, se volvería a ejecutar la instrucción anterior y a evaluar la condición, etc., hasta que en algún momento, por fin, el resultado fuera TRUE.

Por tanto, la primera vez que se ejecuta la instrucción N:=N+1, N, que valía 0, pasa a valer 1. Tras eso, el resultado de la expresión N > 100 es, por supuesto, FALSE, con lo que se volvería a ejecutar N:=N+1 (N valdrá entonces 2), a evaluar N > 100, etc.

Por fin, llegará un momento en que N valdrá 100; tras ejecutarse N:=N+1 valdrá 101 y entonces al evaluarse N > 100 el resultado será TRUE, con lo que se pasará a la instrucción WRITELN que hay a continuación. Traduciendo del inglés resulta todo muy claro:

REPEAT	N:=N+1	UNTIL	N > 100;
“Repetir	N:=N+1	hasta que	N mayor que 100“

La instrucción a repetir puede ser cualquiera, simple o estructurada.

Cuando la instrucción es una secuencia, como aquí hay dos palabras reservadas, REPEAT y UNTIL, por delante y detrás de ella, no hace falta poner BEGIN y END para delimitarla. Por tanto, la o las instrucciones a repetir se escriben una detrás de otra separadas por punto y coma entre las palabras REPEAT y UNTIL. Si en el programa anterior cambiáramos la estructura REPEAT a:

```
REPEAT
 WRITELN (N);
 N:=N+1
UNTIL  N > 100;
```

en la pantalla aparecerían 0, 1, 2, ... hasta 100.

A las situaciones en que un conjunto de instrucciones se repite una y otra vez se les llama «bucles de programa».

La condición de salida del bucle (o sea, la que decide si hay que seguir repitiendo o no) puede ser cualquier expresión que dé un resultado de tipo BOOLEAN.

El conjunto formado por la palabra REPEAT, la o las instrucciones, la palabra UNTIL y la expresión lógica es a su vez una instrucción estructurada y, por tanto, se le aplica todo lo dicho hasta el momento sobre éstas.

Podemos ahora perfeccionar el programa TRES de este capítulo:

```
PROGRAM TRESMEJOR ;

CONST
  ULTIMOPISO = 15 ; ULTIMALETRA = 'E';
VAR
  PISO : 1..ULTIMOPISO;
  LETRA: 'A'..ULTIMALETRA;
  VALE : BOOLEAN ;
  I    : INTEGER;
  C    : CHAR;

BEGIN
  (* Repetir la petición del piso hasta que sea correcto: *)
  REPEAT
    WRITE ('Introduzca el piso: '); READLN (I);
    VALE := (I > 0)  AND  (I <= ULTIMOPISO);
    IF NOT VALE THEN WRITELN ('Piso incorrecto.')
                ELSE PISO := I  (* si es correcto, lo guarda *)
  UNTIL  VALE;

  (* Lo mismo con la letra: *)
  REPEAT
    WRITELN ('Introduzca letra: '); READLN (C);
    VALE := (C >= 'A')  AND  (C <= ULTIMALETRA);
    IF NOT VALE THEN WRITELN ('Letra incorrecta.' )
                ELSE LETRA := C
  UNTIL VALE;

  WRITELN ('El piso introducido es el ',PISO,'-',LETRA)
END.
```

Gracias a las estructuras REPEAT, se pedirán el piso y la letra una y otra vez, hasta que sean correctos.

Nótese el uso de la variable VALE; como la corrección del piso y letra hay que analizarla para avisar en su caso (estructura IF), guardamos el re-

sultado de la prueba en VALE para así no tener que repetir la comprobación tras UNTIL.

Hay que tener en cuenta dos importantes aspectos de la estructura REPEAT:

1. Si el bloque de instrucciones a repetir no afectara para nada a la condición de salida, podríamos tener un «bucle infinito»:

```
PROGRAM CUATROMALO1;
  VAR N ,J : INTEGER;
BEGIN
 N:=0;
 J:=0;

 REPEAT
   WRITELN (N);
   N:=N+1
 UNTIL  J=100;

 WRITELN ('Se acabó.')
END.
```

La condición de salida (J=100) nunca se cumple y por más que se repitan una y otra vez las dos instrucciones del bucle, nunca se cumplirá, con lo que la ejecución del programa no avanzará nunca: tenemos un bucle infinito. (Por cierto, si en algún momento se deseara tener un bucle infinito deliberadamente, lo mejor es escribir **REPEAT ... UNTIL FALSE.**)

2. La condición de salida (la expresión **BOOLEAN**) se evalúa tras la ejecución de la o las instrucciones del bucle. Por ello, aunque su valor fuera **TRUE** ya desde un principio, siempre se ejecutarán las instrucciones al menos una vez:

```
PROGRAM CUATROMALO2;
  VAR N: INTEGER;
BEGIN
 N:= 1000;

 REPEAT
   WRITELN (N);
   N:=N+1
 UNTIL  N > 100;

 WRITELN ('Se acabó.')
END.
```

Este programa escribirá 1000 en pantalla.

Para los casos en que se deba comprobar la condición de salida ANTES de ejecutar las instrucciones del bucle se dispone de la estructura WHILE, que veremos a continuación.



LA ESTRUCTURA WHILE

Esta estructura es similar a la instrucción **REPEAT**, con dos diferencias:

1. La expresión que controla si se debe seguir con las repeticiones o no, se evalúa ANTES de ejecutar las instrucciones del bucle.
2. Se sale del bucle cuando la condición deja de cumplirse, es decir, cuando el resultado de la expresión de control es **FALSE**.

Veamos un ejemplo:

```
PROGRAM CINCO;
 VAR N: INTEGER;
BEGIN
 N:=0;
WHILE  N <= 100  DO
  BEGIN
    WRITELN (N);
    N:= N+1
  END;  (* Aquí se acaba la estructura WHILE *)

WRITELN ('Se acabó.')
END.
```

Una vez más, la traducción del inglés resulta clarificadora:

WHILE	N <= 100	DO	...
"Mientras	N menor o igual que 100	hacer	(la instrucción)

Es decir, entre las palabras reservadas **WHILE** y **DO** se escribe la condición de control (una expresión cuyo resultado sea de tipo **BOOLEAN**) y tras todo ello, la instrucción a repetir, que puede ser cualquiera, estructurada o no (en el ejemplo es una secuencia).

Durante la ejecución del programa, al llegarse a una estructura **WHILE** se evalúa la condición, y si el resultado fuera **TRUE** se pasaría a ejecutar la instrucción a repetir. Tras ello, se volvería a evaluar la condición y

a ejecutar la instrucción caso de que el resultado hubiera sido nuevamente TRUE, etc., hasta que llegara un momento en que el resultado fuera FALSE, en cuyo caso se continuaría la ejecución del programa con la instrucción que hubiera a continuación de la estructura.

Por ello, si en el ejemplo se cambiara la instrucción N:=0 por N:=1000, nunca se llegaría a ejecutar el bucle, pues ya la primera vez el resultado de la condición sería FALSE.

Nuevamente la estructura en su conjunto es toda ella una instrucción estructurada.

Nota: Si se desea profundizar en estas cuestiones se puede observar que la instrucción WHILE equivale a una combinación de IF y REPEAT:

```
WHILE condición DO instrucción
```

(«mientras se dé la condición, haz la instrucción») equivale a:

```
IF  condición THEN
    REPEAT
       instrucción
    UNTIL NOT condición
```

(o sea, «si se da la condición, repite la instrucción hasta que no se dé»).

Para terminar, veamos un programa que calcula y presenta las potencias de dos, menores que un número dado.

```
PROGRAM  POT2;
  VAR  N, TOPE :INTEGER; ERROR, HAYMARGEN: BOOLEAN;
BEGIN
  N:=2;          .        (* 2 es la primera potencia *)
  (* Pedir límite hasta que sea correcto: *)
  REPEAT
    WRITE ('Límite: '); READLN (TOPE);
    ERROR := (TOPE < 0);     (* Error si límite negativo *)
    IF ERROR THEN WRITELN ('No vale. Repita.')
  UNTIL NOT ERROR;

  HAYMARGEN:=TRUE;
  WHILE (N <= TOPE) AND HAYMARGEN DO
    BEGIN
```

```
        WRITELN (N:6);
        HAYMARGEN:= (N <= MAXINT DIV 2);
        IF HAYMARGEN THEN N:=N*2
        (* cada potencia es igual al doble de la anterior *)
       END;

     WRITELN ('Se acabó.')
   END.
```

Se ha utilizado la variable HAYMARGEN para evitar calcular un valor que se salga de los límites admitidos para los números enteros.

LA ESTRUCTURA FOR

En el programa CUATRO de este capítulo se repetía una instrucción un número preestablecido de veces por medio de una estructura REPEAT y la utilización de la variable N como "contador de programa".

Es tan habitual tener que repetir instrucciones o bloques de instrucciones un número fijo de veces, que el PASCAL dispone de una estructura específica para ello: la estructura FOR.

Con ella el programa CUATRO (en la versión que escribía los diferentes valores de N) quedaría:

```
PROGRAM CUATROCONFOR;
 VAR  N: INTEGER;
BEGIN
 FOR N:=0 TO 100 DO  WRITELN (N);

 WRITELN ('Se acabó.')
END.
```

La estructura es la siguiente:

FOR (contador) := (primer valor) TO (último valor) DO (instrucción)

Tras la palabra reservada FOR, se escribe la expresión de asignación del primer valor al contador de programa; tras ello siguen la palabra re-

servada TO, el último valor que tomará el contador, la palabra DO y, por fin, la instrucción a repetir, simple o estructurada.

Al ejecutarse el programa, cuando se llega a una estructura FOR, al contador de programa se le asigna el primer valor. Tras ello se ejecuta la instrucción y el contador pasa a tomar el siguiente valor de manera automática, volviéndose a ejecutar la instrucción nuevamente y a variar el contador, etc., hasta que, por fin, la instrucción se ejecuta teniendo el contador el valor especificado como último, momento tras el cual se pasa a ejecutar lo siguiente a la estructura.

Traduciendo del inglés:

FOR	N:=0	TO	100	DO	WRITELN (N)
"Para	N igual 0	hasta	100	hacer	WRITELN (N)"

Para especificar los valores inicial y final se puede utilizar cualquier expresión que proporcione un resultado del mismo tipo que la variable utilizada como contador.

Aspectos importantes:

— El valor final se calcula al llegarse a la estructura FOR; por ello, si alguno de los elementos de la segunda expresión cambiara durante alguna de las repeticiones del bucle, el valor final no se vería afectado. Por ejemplo, si N y TOPE son variables INTEGER:

```
TOPE:= 10;
FOR N:=0 TO TOPE DO  TOPE:=TOPE +1
```

TOPE pasará de valer 10 a valer 11, 12, etc., pero la instrucción del bucle sólo se repetirá para N valiendo desde 0 hasta 10.

— Si el valor final fuera anterior al inicial, no se ejecutarían ni una vez las instrucciones del bucle:

```
TOPE :=5;
FOR N:=10 TO TOPE DO WRITELN (N);
```

Así, N no se llega a escribir nunca (es decir, la comprobación de llegada al valor final se hace antes de ejecutarse la instrucción).

La variable de control no puede ser de cualquier tipo. Sólo el tipo INTEGER y aquellos otros que tienen asociados números ordinales son ad-

misibles (junto con sus posibles subrangos), es decir, aquellos tipos cuyos posibles valores están ordenados y con los que se pueden usar las funciones PRED, SUCC y ORD. Por ahora sólo hemos visto el tipo CHAR (y el BOOLEAN). Podríamos, por tanto, escribir:

```
PROGRAM ABC;
  VAR C:CHAR;
BEGIN
  FOR C:= 'A' TO 'Z' DO WRITE (C)
END.
```

Este programa nos escribiría el abecedario.

Existe una forma alternativa de la estructura FOR para cuando se desea que la variable de control vaya cambiando de valor en orden inverso al normal. Consiste en usar la palabra reservada DOWNTO en lugar de TO.

En el programa CUATROCONFOR podríamos poner:

```
FOR N := 100 DOWNTO 0 DO WRITELN (N)
```

y se escribiría en pantalla 100, 99, 98, ... hasta 0.

La traducción del inglés sería :

«Para N igual 100 para abajo hasta 0 hacer ...»

El segundo aspecto antes mencionado es aquí el inverso. Con:

```
FOR N:=0 DOWNTO 10 DO WRITELN (N)
```

no se llegaría a escribir nada.

Igualmente podríamos hacer con el tipo CHAR:

```
PROGRAM CBA;
  VAR C: CHAR;
BEGIN
 FOR C:= 'Z' DOWNTO 'A' DO WRITE (C)
END.
```

La estructura FOR es a su vez una instrucción estructurada y, por tanto, podría formar parte de la instrucción a repetir en otra estructura FOR:

```
PROGRAM DOBLEFOR;
 (* Este programa saca todas las combinaciones de piso y letra *)
CONST
  ULTIMOPISO = 15;
  ULTIMALETRA = 'E';
VAR
  PISO :INTEGER;
  LETRA : CHAR;
BEGIN
 FOR PISO :=1 TO ULTIMOPISO DO
  BEGIN
   WRITELN;
   WRITELN('En el piso ',PISO,' tenemos: ');
   FOR LETRA:= 'A' TO ULTIMALETRA DO WRITE  (PISO:4,'-',LETRA);
   WRITELN
  END
END.
```

Por último, dos advertencias y un consejo dirigidos fundamentalmente a los amantes del BASIC:

— No se debe cambiar el valor de la variable de control mediante una instrucción de asignación (o READ o READLN) dentro del bucle; podrían suceder cosas imprevistas. Por tanto, NO se deben hacer cosas como:

```
FOR N:=0 TO 100 DO
 BEGIN
  .....
  N:=117
 END
```

— El valor del contador de programa al acabar de ejecutarse la estructura FOR puede que sea el siguiente al especificado como final, pero puede que no; depende de cada compilador en concreto.

— El único tipo de contador numérico admitido es el INTEGER, y además sólo se admiten incrementos de 1 (o de -1 con DOWNTO). Por ello, lo mejor para cuando se desean variaciones distintas de la unidad es utilizar una estructura REPEAT o WHILE:

```
R:=PRIMERVALOR
REPEAT  ... ; ... ; R:=R+INCREMENTO  UNTIL R > ULTIMOVALOR;
```

DISEÑO DE UN PROGRAMA 5

AHORA que ya conocemos las principales estructuras de control del PASCAL, resulta posible escribir programas que hagan tareas más complicadas.

Como ejemplo de ello, vamos a desarrollar uno para calcular todas las posibles maneras en que un número entero puede ser descompuesto como producto de otros dos. Por ejemplo, el número 60 puede descomponerse como 2 por 30, 3 por 20, 4 por 15, etc.

En primer lugar, si deseamos descomponer no uno, sino varios números, tendremos que utilizar alguna estructura de bucle para poder repetir los cálculos varias veces. Podríamos escribir el programa para que procese una cantidad fija de números, en cuyo caso utilizaríamos una estructura FOR, pero parece más útil poder procesar todos los que se quiera hasta que se indique al ordenador que ya no se desea seguir adelante.

Para indicar esto, una forma muy sencilla podría ser dar el número cero cuando se pida un nuevo número para procesar. Entonces tendríamos que usar estructuras como

```
WHILE NUMERO<>0 DO ...      o
REPEAT ... UNTIL NUMERO=0
```

Como ya se vio en su momento, en la instrucción REPEAT la comprobación para salir del bucle se hace después de ejecutarlo; en nuestro caso no deseamos procesar el número cero, pues no se puede descomponer de ninguna manera y, por tanto, usaremos una estructura WHILE.

Estamos ya en condiciones de escribir un boceto del programa:

1. Leer primer número.
2. Mientras (while) el número sea distinto de cero hacer:
   — Procesar y sacar resultados.
   — Leer siguiente número.
3. Mostrar en pantalla mensaje de despedida.

Resulta evidente cómo programar los puntos 1 y 3, pero, sin embargo, lo de «procesar» en el punto 2 es más complejo.

Decir que, por ejemplo, 60 se puede descomponer como 5 multiplicado por otro número entero equivale a decir que si dividimos 60 por 5 el resultado exacto es precisamente ese número entero (así, como 60 entre 8 es 7,5, 60 no se puede descomponer como 8 por otro entero).

Por tanto, una forma de obtener todas las posibles descomposiciones es probar para todos los factores posibles si la división exacta da un número entero o, en otras palabras, probar si al hacer la división sin decimales el resto sale cero. El factor más pequeño a considerar es el 2, cuyo factor correspondiente (el mayor posible) será NUMERO DIV 2.

Puede resultar que el número en cuestión no se pueda descomponer como producto de ningún par de factores enteros; en ese caso se dice que el número es PRIMO. Podemos entonces preparar el programa para que avise cuando el número sea primo. Para ello lo más fácil es anotar al principio en algún sitio que el número es primo, y si después se encuentra alguna manera de descomponerlo, se tacha la anotación; si, tras haber probado todos los posibles factores, la anotación permaneciera intacta, el número sería primo.

Un boceto del contenido del bucle WHILE podría ser, por tanto:

2A. Anotar qué NUMERO es primo.

2B. Para FACTOR valiendo desde 2 hasta NUMERO DIV 2 hacer:

   Si NUMERO dividido por FACTOR es un número entero, entonces:

   — Presentar FACTOR y NUMERO DIV FACTOR.
   — Tachar la anotación de que NUMERO es primo.

2C. Si todavía está la nota, avisar que NUMERO es primo.

2D. Leer siguiente número.

Resulta evidente la presencia de estructuras FOR e IF.

Podemos pasar ya a escribir el programa en PASCAL:

```
PROGRAM FACTORES;
(* Programa para descomponer un número en dos factores *)
```

```
VAR  NUMERO,FACTOR: INTEGER;
     PRIMO        : BOOLEAN;

BEGIN
 WRITELN ('Introduzca número a descomponer (0 para acabar).');
 READLN (NUMERO); NUMERO:=ABS(NUMERO);              (* punto 1 *)
 WHILE (NUMERO<>0) DO                               (* punto 2 *)
   BEGIN
     PRIMO:=TRUE;                                   (* punto A *)
     FOR FACTOR:=2 TO NUMERO DIV 2 DO               (* punto B *)
       IF (NUMERO MOD FACTOR = 0) THEN  (* Si el resto es 0 *)
         BEGIN
           WRITELN (FACTOR:5,'  por ',NUMERO DIV FACTOR:6);
           PRIMO:=FALSE
         END;
     IF PRIMO THEN WRITELN ('Este número es primo.');  (* C *)
     WRITELN;
     WRITELN ('Introduzca número a descomponer.');
     READLN (NUMERO); NUMERO:=ABS(NUMERO)              (* D *)
   END;
 WRITELN ('ADIOS, HASTA LA PROXIMA.')               (* punto 3 *)
END.
```

En el programa se ha tomado la precaución de hacer que NUMERO sea siempre positivo (NUMERO:=ABS(NUMERO)) pues si, por ejemplo, valiera -12, tendríamos la instrucción:

FOR FACTOR:=2 TO -6 DO ...

que, como sabemos, hace que el bucle no se ejecute ni una vez, pues habría que usar DOWNTO en lugar de TO.

El desarrollo que se ha hecho de este programa, descomponiéndolo en tareas más sencillas, éstas a su vez en otras, y así hasta llegar a unas que sean fácilmente convertibles en instrucciones de programa, hace que la posibilidad de error disminuya mucho. No obstante, es necesario probar el programa para garantizar su correcto funcionamiento; siempre queda la posibilidad de que casualmente aquello que no hayamos probado sea lo que falle, pero escogiendo adecuadamente las pruebas se puede hacer que esa probabilidad sea pequeña.

Así, conviene probar el programa para los casos un poco especiales como cuando NUMERO vale 0, en que no se calcula nada y se acaba la ejecución del programa, 1,2 y 3, en que el bucle FOR no se ejecuta ni una vez, algún número negativo, etc.

El hecho de que este programa detecte cuando un número es primo sugiere su adaptación para obtener todos los números primos hasta un máximo dado. Un boceto podría ser:

1. Leer MAXIMO.
2. Para NUMERO valiendo desde 1 hasta MAXIMO hacer:
   - Procesar.
   - Si NUMERO es primo, mostrarlo.
3. Mostrar en pantalla mensaje de despedida.

«Procesar» sería similar al anterior pero sin mostrar los factores. El programa definitivo quedaría:

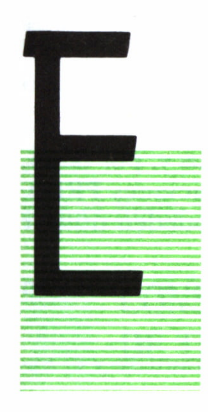
```
PROGRAM PRIMOS;
(* Programa para obtener números primos *)

VAR  MAXIMO,NUMERO,FACTOR: INTEGER;
     PRIMO          : BOOLEAN;

BEGIN
 WRITELN ('Introduzca limite (máximo=',MAXINT,').');
 READLN (MAXIMO); MAXIMO:=ABS(MAXIMO);
 FOR NUMERO:=1 TO MAXIMO DO
   BEGIN
     PRIMO:=TRUE;
     FOR FACTOR:=2 TO NUMERO DIV 2 DO
       IF (NUMERO MOD FACTOR = 0) THEN  PRIMO:=FALSE;
     IF PRIMO THEN WRITE (NUMERO:8);
   END;
 WRITELN;
 WRITELN ('ADIOS, HASTA LA PROXIMA.')
END.
```

A medida que los números van siendo mayores, el tiempo para procesarlos aumenta rápidamente. Este método de obtención de primos es poco eficiente; existen otros métodos mucho más eficaces (e incluso éste mismo es posible mejorarlo bastante), pero todo eso escapa del alcance de este libro.

EL TIPO REAL 6



L tipo **REAL** es el utilizado para manejar números que puedan ser no enteros o fuera de los límites establecidos para los de tipo **INTEGER**.

Siempre que tratemos con números muy grandes o con decimales habrá que acudir, pues, al tipo **REAL**.

En los demás casos conviene utilizar el tipo **INTEGER** pues precisa menor cantidad de memoria y se procesa a mayor velocidad.



PRECISION Y MAGNITUD DE LOS NUMEROS DE TIPO REAL

Los números reales se guardan en memoria descomponiéndolos en dos números sin decimales.

Por ejemplo, como 123400000 es igual que 0,1234 multiplicado por 10 elevado a 9 (o sea, por 1000000000, un 1 y 9 ceros), podríamos guardar por un lado 1234 y por otro 9.

1234 es lo que se llama la **MANTISA** y 9 el **EXPONENTE**.

Más ejemplos:

–395,37 es igual a –0,39537 por 1000, luego se guardaría –39537 y 3

0,000734 es igual a 0,734 dividido entre 1000; en este caso se guardaría 734 y –3, indicando el exponente negativo que lo que hay que hacer para obtener el número es dividir la mantisa por un 1 y tantos ceros como indique el valor absoluto del exponente.

Es decir, se corre la coma tantos lugares como sea necesario para que el valor absoluto del número quede lo más próximo posible a 1 pero por debajo, y a eso se le llama mantisa. Al número de posiciones que se haya tenido que correr se le llama exponente, indicando su signo para dónde hubo que correr la coma.

En la práctica, la forma de guardar los números de tipo REAL es ligeramente distinta pues los ordenadores utilizan números binarios, esto es, con ceros y unos solamente, pero el fundamento es el mismo. Por supuesto, todo esto se hace de manera automática sin que nosotros nos enteremos. A este sistema de almacenamiento en memoria se le denomina de «coma flotante».

Como consecuencia de esta forma de guardar los números de tipo REAL suceden varias cosas:

1. Cada número REAL se descompone en DOS números y, por tanto, no pueden tener números ordinales asociados. Así, las funciones PRED, SUCC y ORD no se pueden aplicar a valores REAL, ni se pueden utilizar variables REAL para controlar bucles FOR.

No obstante, SI es posible comparar valores REAL como se hacía con los INTEGER.

2. Los dos números en que se descomponen, o sea, la mantisa y el exponente, tienen límites que dependen de cada compilador:

— La limitación de la mantisa supone una disminución de la precisión con que se pueden guardar los números. Si pretendiéramos guardar, por ejemplo, el resultado de dividir 100 entre 3, que vale 33,333... con infinitos treses, en la memoria quedaría:

33,3333333... = 0,3333333... por 100

ahora bien, si la mantisa pudiera tener como máximo 4 cifras, sólo podríamos guardar 3333 y 2 que, en realidad, es lo que corresponde a 33,33.

Esto es lo que se denomina error de redondeo. Así, si dividiéramos 100 entre 3 y lo multiplicáramos luego por 3 obtendríamos 99,99 en lugar de recuperar el valor original de 100, que es lo esperable en principio.

Esta situación, debido al uso de números binarios, se puede dar con números que aparentemente no necesiten muchas cifras, pues, por ejemplo, aunque 0,2 sólo necesita una cifra, en binario 0,2 se escribe como 0,001100110011...

Por todo esto, es peligroso comparar dos números reales para ver su igualdad, pues debido al error de redondeo, números que esperábamos que fueran iguales pueden resultar distintos.

Normalmente se pueden esperar al menos 7 u 8 cifras de precisión. Esto se debe consultar en el Manual.

— La limitación del exponente supone, por un lado, una límite para la magnitud de los números REAL que se puedan guardar en memoria. Por ello, si el máximo exponente fuera 37 (caso corriente), el número 1273400...(hasta 50 ceros), igual a 0,12734 por 10 elevado a 55, estaría fuera de rango.

Por otro lado, supone que los números muy pequeños se redondearán a cero al ser guardados en memoria. Por ejemplo:

0,00...(50 ceros)...1254 es igual a 0,1254 entre 10 elevado a 50

y si el límite inferior fuera −37, el número se guardaría como un cero.



CONSTANTES Y VARIABLES DE TIPO REAL

Como se puede imaginar, las constantes de tipo REAL deben ser claramente diferenciables de las de tipo INTEGER para que el compilador sepa cómo guardarlas en memoria.

Si para utilizar en expresiones de tipo REAL necesitáramos un valor, por ejemplo, 15, no podríamos escribirlo tal cual, pues parecería INTEGER.

Para distinguirse, las constantes REAL deben tener punto decimal o exponente o ambos (en inglés, para separar las cifras decimales de las demás se utiliza un punto en lugar de una coma). El exponente se separa del resto del número por medio de la letra E. Veamos unos ejemplos:

15.0	DEBE haber cifras a ambos lados del punto.
3E3	3E3 significa 3 por 10 elevado a 3, o sea, 3000.
−3.17E-4	Equivale a −3,17 entre 10000, o sea, −0,000317.
0.3333333333333	Como ya hemos mencionado, probablemente no se utilizarán todas la cifras.

Las constantes REAL, como las INTEGER, pueden declararse al principio del programa:

```
CONST PI = 3.14159265;
```

Esta constante en concreto se encuentra ya definida con este mismo nombre en muchas versiones de PASCAL. También las variables se definen como las INTEGER, pero usando la palabra reservada REAL:

```
VAR ANGULO, PESO: REAL;
```

Sin embargo, y debido a la carencia de ordinales, no se pueden definir subrangos.

La asignación de datos a una variable se hace de manera análoga:

```
ANGULO := PI;
PESO := 60.0;
```

Si escribiéramos, por ejemplo, PESO:= 60, al ser 60 una constante de tipo INTEGER, como la variable de destino es REAL se tiene que producir una transformación de la forma en que se guarda el número 60 (sin que nosotros nos apercibamos de ello). Esto se lleva tiempo de cálculo, por lo que es conveniente ser metódico y escribir constantes reales siempre que corresponda.



EXPRESIONES REAL

Las expresiones de tipo REAL son aquéllas en que aparecen constantes, predeclaradas o no, y variables de tipo REAL combinadas entre sí mediante los operadores + , – , * y / .

El operador DIV sólo es aplicable al tipo INTEGER, pero en su lugar aquí se tiene el operador / para hacer divisiones, ahora sí, con decimales.

El orden de cálculo es similar al de las expresiones INTEGER.

100.0 * (37.0 – 4.5) / 2.0 dará el resultado REAL 1625.0.

En una expresión REAL pueden aparecer constantes o variables INTEGER, cuyo valor es convertido a tipo REAL automáticamente antes de ser utilizado.

En expresiones como:

3 + I * I / 3

donde I fuera una variable o constante entera, aunque todos los términos son INTEGER , al descubrirse durante el cálculo el operador / , como es exclusivo de los REAL, se pasa a utilizar el tipo REAL, y el resultado será REAL.

Este cambio sobre la marcha puede dar lugar a situaciones curiosas que deben ser analizadas con cuidado. Por ejemplo, la expresión:

1000 * 1000 / 10000

aunque su resultado es 100, como se empieza a calcular por la izquierda, produce un error, pues 1000 (INTEGER) por 1000 (INTEGER) da un valor fuera de rango, ya que todavía no se ha descubierto que es una expresión REAL. Si escribiéramos :

1000 / 10000 * 1000 o 1000 * 1000.0 / 10000 etc.

se resolvería el problema, pues antes de hacer ninguna operación se pasaría a utilizar el tipo REAL.



CONVERSION DE VALORES REAL A INTEGER

Ya se ha visto que la conversión de INTEGER a REAL se produce de manera automática; por ello, si I fuera una variable o constante INTEGER cuyo valor quisiésemos guardar en la variable REAL R, bastaría con escribir R:=I.

Sin embargo, a la inversa hay que definirlo de manera explícita para así poder elegir entre dos posibles modos de conversión:

— La función ROUND (R), donde R es una variable, constante o expresión REAL, da un resultado INTEGER lo más próximo posible al valor de R:

ROUND (14.28) equivale a poner 14 y
ROUND (14.59) equivale a poner 15,

de ahí su nombre ("redondea").

— La función TRUNC (R) da un resultado INTEGER igual al número entero más cercano por debajo:

TRUNC (7.3) es igual a 7 y
TRUNC (7.99) también.

Por supuesto, si el valor resultante estuviera fuera de los límites INTEGER admisibles se produciría un error.

Nótese que ROUND (R) es igual a TRUNC (R+0.5).

Si quisiéramos calcular la división de dos variables REAL sin obtener decimales y guardar el resultado en una variable INTEGER escribiríamos:

I:= TRUNC (A/B) o bien I:= ROUND(A) DIV ROUND(B)

(estas expresiones no siempre darán el mismo resultado).



FUNCIONES DE LIBRERIA

El PASCAL proporciona varias funciones matemáticas predefinidas. Como tales funciones, el parámetro o valor del cual se quiere obtener la función se debe escribir entre paréntesis a continuación del nombre de la función. Devuelven un valor de tipo REAL y, por tanto, se pueden incor-

porar a cualquier expresión de este tipo. Por supuesto, el parámetro puede ser una expresión REAL cualquiera. Son las siguientes:

FUNCION	SIGNIFICADO		ejemplos
(*) ABS	valor absoluto	ABS (-3.5)	es igual a 3.5
(*) SQR	cuadrado	SQR (2.0)	es igual a 4.0
SQRT	raíz cuadrada	SQRT (6.25)	es igual a 2.5
EXP	exponencial	EXP (1.0)	es igual a 2.718281
LN	logaritmo natural	LN (1.0)	es igual a 0.0
SIN	seno	SIN (PI/2.0)	es igual a 1.0
COS	coseno	COS (PI)	es igual a -1.0
ARCTAN	arcotangente	ARCTAN (1.0)	es igual a PI/4.0

Se pueden producir errores durante el cálculo de algunas de estas funciones si el valor del parámetro se sale de los límites establecidos para cada función:

SQRT (-2.0) no es calculable.

Una protección en este caso sería escribir SQRT (ABS(R)), donde R fuera la variable, constante o expresión REAL cuya raíz cuadrada se quisiera obtener.

El parámetro puede ser de tipo INTEGER, pues éste se convertirá automáticamente en REAL, excepto en las funciones marcadas con (*), que proporcionan un resultado del mismo tipo que el parámetro.



EXPRESIONES CON VALORES REAL QUE DAN RESULTADO BOOLEAN

Al igual que con el tipo INTEGER, éstas son las comparaciones:

> , < , >= , <= , = , <>

Debe tenerse en cuenta el comentario anterior sobre las comparaciones de igualdad y desigualdad:

100.0 / 3.0 * 3.0 = 100.0

podría dar como resultado FALSE (y usando <> podría dar TRUE).

Una forma de arreglar esto puede ser poner como condición de igualdad de dos valores el que su diferencia sea menor que una cierta cantidad:

```
ABS (A-B) < DIFERENCIAMAXIMA
```

donde A y B son los valores a comparar.

Normalmente los compiladores admiten comparar valores de tipo REAL con valores de tipo INTEGER, pues éstos son convertidos a REAL antes de hacer la comparación.

A diferencia del tipo INTEGER, y debido a la carencia de ordinales, no existe la función ODD para el tipo REAL.



ENTRADA Y SALIDA DE DATOS DE TIPO REAL

Las variables REAL pueden tomar el valor que se introduzca desde teclado por medio de las instrucciones READ y READLN, que se utilizan de la misma manera que con las variables INTEGER. Valores tecleados aceptables son:

```
3        72.7        0.727E2        727E-1
```

Como ejemplo veamos una forma de evitar que al leer una variable de tipo INTEGER se pueda producir un error por salida de rango. Utilizaremos una variable REAL para recoger el número tecleado y sólo lo transferiremos a la de tipo INTEGER tras haber comprobado que tiene un valor aceptable:

```
PROGRAM UNO;
  VAR    AUXILIAR: REAL;
         I       : INTEGER;
         VALE    : BOOLEAN;
BEGIN
 REPEAT
   WRITE ('Valor = '); READLN (AUXILIAR);
   (* mirar si el entero correspondiente es lícito: *)
   VALE := (ABS (AUXILIAR) <= MAXINT);
```

```
    IF NOT VALE THEN WRITELN ('No vale. Repita.')
  UNTIL VALE;
  I:= ROUND (AUXILIAR);
END.
```

Por supuesto, también sería posible teclear un número fuera de rango incluso para una variable REAL, pero es poco probable que se hiciera por equivocación.

La presentación por pantalla se hace igual que con los valores INTEGER. Los valores REAL se presentan en notación exponencial, esto es, escribiendo mantisa y exponente separados por la letra E. No obstante, es posible presentarlos con aspecto «normal». Para ello, a continuación de la expresión REAL se indica el número de espacios a ocupar y el número de cifras decimales a presentar separados por dos puntos. Por ejemplo:

```
WRITELN ( 27.2 , 27.2:6:2, 27.2:10:4 )
```

produciría algo como:

```
2.72000000E+01  27.20     27.2000
```



PROGRAMA DE EJEMPLO

El programa de ejemplo que se hizo en el capítulo anterior tenía, entre otros, el inconveniente de que podía presentar parejas de factores repetidas. Así, el número 16 se podría descomponer, según ese programa, en 2 por 8, 4 por 4 y 8 por 2, sin tener en cuenta que, realmente, 2 por 8 es la misma descomposición que 8 por 2.

Para evitar esto, habría que explorar sólo las parejas en que el primer número fuera menor o igual que el segundo.

Dado un número cualquiera, la pareja de factores en que los dos son iguales es aquélla en que éstos son la raíz cuadrada del número. Puede que ésta no sea entera, pero lo que está claro es que si hay algún par de factores en que se pueda descomponer el número y uno de ellos es menor que la raíz cuadrada, el otro ha de ser forzosamente mayor. Por ejemplo,

como la raíz cuadrada de 64 es 8, si uno de los factores fuera menor o igual que ella el otro sería mayor o igual:

2 por 32, 4 por 16, 8 por 8, y, desde ahí, 16 por 4 y 32 por 2.

Por tanto, el límite de exploración del primer factor debiera ser la raíz cuadrada del número a descomponer.

Por otra parte, se puede comprobar que cualquier número entero se puede expresar como 6 por N, más 0, 1, 2, ó 3, o menos 1 ó 2, donde N es otro número entero. Por ejemplo:

12	es igual a	6 por 2	más	0
13	es igual a	6 por 2	más	1
14	es igual a	6 por 2	más	2
15	es igual a	6 por 2	más	3
16	es igual a	6 por 3	menos	2
17	es igual a	6 por 3	menos	1
18	es igual a	6 por 3	más	0
19	es igual a	6 por 3	más	1

Los números que se pueden expresar como 6 por N más 0 ó 2 o menos 2 son divisibles por 2 (y la mitad será 3 por N más 0 ó 1 o menos 1).

Los números que se pueden expresar como 6 por N más 0 ó 3 son divisibles por 3 (y la tercera parte será 2 por N más 0 ó 1)

Por ello, a la hora de explorar posibles números primos, una forma de ganar tiempo es probar sólo aquéllos que se puedan expresar como 6 por N más o menos 1, con lo cual se evita probar números que sean divisibles por 2 ó 3.

Con todos estos cambios, el programa de obtención de primos quedaría:

```
PROGRAM PRIMOSMEJOR;
(* programa para obtener números primos: *)
VAR
 MAXIMO,NUMERO,FACTOR,N: INTEGER;
 PRIMO,ULTIMOCONMAS: BOOLEAN;
BEGIN
 WRITELN ('Introduzca limite (máximo=',MAXINT,').');
 READLN (MAXIMO); MAXIMO:=ABS(MAXIMO);
```

```
WRITE (1 :8, 2 :8, 3 :8);  (* escribir los primeros primos *)
NUMERO:=5;                 (* primer número a probar *)
N:= 1;
ULTIMOCONMAS:=FALSE;

WHILE NUMERO <= MAXIMO DO
  BEGIN
    PRIMO:=TRUE;
    (* no probar con factor 2,3 o 4 pues no hace falta *)
    FOR FACTOR:=5 TO ROUND(SQRT(NUMERO)) DO
      IF (NUMERO MOD FACTOR = 0) THEN  PRIMO:=FALSE;
    IF PRIMO THEN WRITE (NUMERO:8);

    (* Obtener siguiente número a probar: *)
    IF ULTIMOCONMAS THEN
       BEGIN
         N:=N+1;  (* pasar al siguiente N y probar con -1 *)
         NUMERO:= 6 * N - 1;
         ULTIMOCONMAS:= FALSE
       END
      ELSE
       BEGIN   (* probar con el mismo N y +1 *)
         NUMERO:= 6 * N + 1;
         ULTIMOCONMAS:= TRUE
       END
  END;

WRITELN;
WRITELN ('ADIOS, HASTA LA PROXIMA.')
END.
```

Realmente, en lugar de ROUND habría que utilizar TRUNC, pero si, por ejemplo, la raíz cuadrada de 25 fuera 4.99999999 debido al error de redondeo, el límite del bucle FOR sería incorrecto.

Se ha utilizado la variable BOOLEAN ULTIMOCONMAS para indicar si el último número probado fue del tipo 6*N+1 o no.

Va tomando alternativamente los valores FALSE y TRUE, por lo que las dos asignaciones dentro de la estructura IF se podrían cambiar por la única ULTIMOCONMAS:= NOT ULTIMOCONMAS a continuación de ella. La modificación del programa de los factores es inmediata.

MAS TIPOS DE DATOS 7



Ya se ha mencionado que, además de los tipos de datos existentes en PASCAL, es posible crear nuestros propios tipos de datos.

La definición de tipos se hace en la zona de descripción de datos y antes de la definición de variables. Va precedida de la palabra reservada TYPE, tras la cual se escriben las descripciones de datos separadas entre sí por punto y coma.

Como los procedimientos READ, READLN, WRITE y WRITELN ya están programados de antemano, no están preparados para ser utilizados con los tipos creados por nosotros.

Vamos a ver algunos de los posibles nuevos tipos.



TIPOS ESCALARES

A menudo se trabaja con datos que no son números ni letras. Por ejemplo, podría ser necesario tener en cuenta el día de la semana para que un programa hiciera o no determinadas cosas.

Para guardar en una variable el día en cuestión, una solución podría ser asociar a cada día de la semana un número: lunes el 1, martes el 2, etc. Entonces podríamos escribir cosas como:

```
IF DIA=7 THEN ...
```

donde DIA sería una variable de tipo INTEGER.

Esto nos obligaría a recordar en cada momento qué números hemos asociado a los posibles valores de nuestros datos.

En PASCAL, sin embargo, es posible crear un nuevo tipo de datos para que se ajuste a nuestras necesidades. Podríamos crear así un tipo (llamémosle DIADELASEMANA) cuyos posibles valores serían LUNES, MARTES, MIERCOLES... al igual que existe el tipo INTEGER cuyos posibles valores son ...-2,-1,0,1,2... o BOOLEAN con TRUE y FALSE.

Definiríamos entonces el tipo enumerando sus posibles valores.

Para ello se escribe, en primer lugar, el nombre del tipo seguido de un igual y de los identificadores de los posibles valores separados entre sí por comas y encerrados entre paréntesis. Los tipos así definidos, junto con los tipos INTEGER, CHAR y BOOLEAN, se llaman *tipos escalares*. Por ejemplo:

```
PROGRAM UNO;
CONST .....; (* Las que hubiera *)
      .....;
TYPE
 DIADELASEMANA=(LUNES,MARTES,MIERCOLES,JUEVES,VIERNES,SABADO,DOMINGO);
 COLORPRIMARIO=(ROJO,VERDE,AZUL);

VAR   HOY,DIA: DIADELASEMANA;
      COLOR: COLORPRIMARIO;
      I,J,K,L: INTEGER;
BEGIN
  ...
END.
```

Con esos tipos así definidos se podría escribir:

```
HOY:=LUNES;
IF (DIA=SABADO) AND (COLOR<>ROJO) THEN ... ;
```

Hay dos restricciones a la hora de crear estos tipos:

— El número de posibles valores está limitado, dependiendo este límite de cada compilador; normalmente es 256.

— Los posibles valores deben ser exclusivos del tipo que hayamos definido. Así, no sería posible crear el tipo VOCALES formado por los valores 'A','E','I','O','U', pues éstos lo son del tipo CHAR.

Como se puede ver, el tipo BOOLEAN sería definible, caso de que no lo estuviera ya, de la siguiente manera:

```
TYPE BOOLEAN = (FALSE,TRUE);
```

Al igual que los tipos CHAR y BOOLEAN, los tipos escalares tienen asociados números ordinales y se encuentran ordenados según éstos.

Estos números se asignan según el orden en que se hayan enumerado los posibles valores al definir el tipo, empezando por cero.

Por ello, es posible utilizar las funciones ORD, PRED y SUCC que dan, respectivamente, el número de orden, el valor anterior y el posterior a uno dado:

ORD(HOY) valdría 1 si HOY valiera MARTES.
PRED(SABADO) valdría VIERNES.
SUCC(COLOR) valdría VERDE si COLOR valiera ROJO.

y comparar valores:

(LUNES < MARTES) valdría TRUE pues ORD(LUNES) es menor que ORD(MARTES)

(Recordemos una vez más que en PASCAL nunca se pueden mezclar tipos.)

Por esto mismo también es posible usar variables de tipo definido por enumeración para control de las estructuras FOR:

```
FOR DIA:=LUNES TO VIERNES DO ...
FOR COLOR:=AZUL DOWNTO ROJO DO ...
```

Para presentar valores de tipos creados por nosotros habría que acudir a artificios como:

```
IF DIA=LUNES THEN WRITE('LUNES')
 ELSE IF DIA=MARTES THEN WRITE('MARTES')
  ELSE ...
```

Y lo mismo para leerlos:

```
WRITELN ('Color? (R,V,A)'); READLN (C); (* C es de tipo CHAR *)
IF C='R' THEN COLOR:=ROJO
 ELSE IF C='V' THEN COLOR:=VERDE
  ELSE ...
```



TIPOS SUBRANGO

Ya se vio en su momento cómo se podía definir una variable de tipo subrango de los tipos CHAR e INTEGER.

Con los otros tipos escalares es posible también definir subrangos. Por ejemplo, estando el tipo DIADELASEMANA ya definido podríamos tener:

```
VAR DIADECURRE: LUNES..VIERNES;
```

con lo que indicamos al compilador que la variable DIADECURRE sólo puede tener valores comprendidos entre LUNES y VIERNES. El subrango se describe, por tanto, poniendo los valores extremos separados por un par de puntos, de manera que el primero sea menor que el segundo. Nunca se podrá definir un subrango del tipo REAL.

Aunque en el ejemplo hemos descrito el subrango al definir la variable, es a veces más conveniente crear primero el tipo subrango y luego utilizarlo al describir las variables:

```
TYPE
  DIADELASEMANA = (LUNES,MARTES,MIERCOLES,JUEVES,VIERNES,
                   SABADO,DOMINGO);
  DIALABORABLE  = LUNES..VIERNES;
  ENTERITO      = 0..9;
  LETRA         = 'A'..'Z';

VAR  DIADECURRE: DIALABORABLE;
     N: ENTERITO;
     C: LETRA;
```

Los tipos subrango en el fondo siguen siendo del tipo que les da origen y por ello **SÍ** se pueden mezclar entre sí y con el tipo escalar asociado si éste es común a ellos. Por ejemplo, si tenemos:

```
VAR A: 1..10; B: 0..30; C: 10..30;
```

el tipo original asociado a **A**, **B** y **C** es **INTEGER** y por ello las instrucciones

```
A:=B; WRITE(B+C); C:=B, etc.,
```

son correctas, aunque al correr el programa se podría producir un error al ejecutar, por ejemplo, **A:=B** si **B** tuviera un valor no aceptable para **A**.

Además de servir para controlar que no se almacenan valores absurdos en las variables, a veces se obtiene un ahorro de memoria, pues el tamaño de la porción necesaria para guardar datos con valores restringidos puede ser menor.

Nota: La mayoría de los compiladores tienen la posibilidad de activar o no la propiedad de controlar si los valores que se guardan en variables de tipo subrango están dentro de los límites permitidos; debe consultarse el Manual para utilizarla.



TIPOS ESTRUCTURADOS

Todos los tipos vistos hasta el momento se refieren a datos simples.

Una variable de tipo **INTEGER** sólo puede tomar un valor en un momento dado y lo mismo sucede con los tipos **CHAR**, **DIADELASEMANA**...

Sin embargo, nos podría interesar guardar en una única variable el peso, la edad, la estatura y el nombre de una persona, o guardar en una sola variable todas las notas de un examen, etc.

En **PASCAL** se pueden definir tipos de datos que a su vez se componen de otros tipos, para así poder tratar con varios valores a la vez; es lo que se llaman *tipos estructurados*.

Vamos a ver a continuación el más común de ellos, el tipo **ARRAY**.



EL TIPO ARRAY

Supongamos por un momento que queremos obtener la media de las notas de un examen y posteriormente obtener la nota más alta. Primero

deberíamos sumarlas todas y dividir el resultado por el número de notas. Haríamos:

```
PROGRAM NOTAS;
VAR
 NOTA1, NOTA2, NOTA3 : REAL;
 SUMA, MEDIA, MAXIMA : REAL;
BEGIN
 WRITELN('INTRODUZCA LAS NOTAS');
 READLN (NOTA1);
 READLN (NOTA2);
 READLN (NOTA3);
 (* Ahora calculamos la media *)
 SUMA:=0.0;
 SUMA:=SUMA+NOTA1; (* Al valor anterior de SUMA le sumamos NOTA1 *)
 SUMA:=SUMA+NOTA2;
 SUMA:=SUMA+NOTA3;
 MEDIA:=SUMA/3.0;
 WRITELN('La nota media es',MEDIA:6:2);
 (* Ahora obtenemos la nota máxima *)
 MAXIMA:=0.0;
 IF NOTA1>MAXIMA THEN MAXIMA:=NOTA1;
 IF NOTA2>MAXIMA THEN MAXIMA:=NOTA2;
 IF NOTA3>MAXIMA THEN MAXIMA:=NOTA3;
 WRITELN('La nota más alta es:',MAXIMA:6:2);
END.
```

Es fácil imaginar cómo sería este programa para calcular la media de cien notas. Si se pudieran guardar todas en memoria como un grupo y no cada una con su propia variable, el programa sería mucho más sencillo.

Las variables de tipo ARRAY permiten guardar en una sola muchos datos del mismo tipo. Es como si tuviéramos una tabla con muchos valores registrados y para escoger uno de ellos dijéramos: «Dame el primer valor de la tabla», o el quinto, o el tercero, etc.

Para utilizar uno de los elementos de la tabla en concreto se necesita, por tanto, indicar cuál de ellos es, por medio de lo que se denomina un INDICE.

Este índice podría ser un valor de tipo INTEGER, con lo que podríamos decir algo como «Guarda esto en el elemento número 5 de la tabla» o «Presenta en pantalla el elemento número 2».

Por tanto, al crear una variable de tipo ARRAY hay que indicar dos cosas:

— Entre qué valores puede estar comprendido el índice. En otras palabras, el subrango al que pertenece éste.

— De qué elementos es la tabla. Podría ser una tabla de números enteros, reales, de caracteres, de variables del tipo DIADELASEMANA...

Para ello se escribe en primer lugar la palabra reservada ARRAY, seguida del subrango al que corresponda el índice puesto entre corchetes; tras ello se escribe la palabra reservada OF, seguida del tipo al que pertenecen los elementos de la tabla. Por ejemplo:

```
VAR NOTAS: ARRAY [1..100] OF INTEGER;
```

crea una variable llamada NOTAS que es una tabla de números de tipo INTEGER. Los elementos de la tabla se escogen indicándolo con un índice que puede estar comprendido entre 1 y 100 (1..100 es un subrango del tipo INTEGER). Tendremos así el elemento número 1 de la tabla, el 2, etc. hasta el 100, o sea, 100 elementos en total.
Al igual que sucedía con el tipo subrango, puede ser más conveniente crear primero:

```
TYPE TIPONOTAS: ARRAY [1..100] OF INTEGER;
VAR  NOTASCLASE1,NOTASCLASE2: TIPONOTAS;
```

Para indicar el elemento de la tabla que se desea utilizar se escribe el nombre de ésta seguido de una constante, variable o expresión que proporcione el valor del índice, puesta entre corchetes:

```
WRITE (NOTAS[1]*2); (* Escribe el doble del elemento 1 de la tabla *)
NOTAS[50*2]:=7;     (* Guarda 7 en el elemento 100 de la tabla *)
```

Los elementos así especificados son del tipo constitutivo del ARRAY. En el ejemplo serían, pues, de tipo INTEGER y, por tanto, se podrían utilizar exactamente igual que cualquier otra variable INTEGER.

Si el ordenador no dispusiera de los corchetes en su juego de caracteres se pueden utilizar en su lugar las parejas (. y .) :

```
NOTAS(.5.):=3;
```

No se pueden comparar variables de tipo ARRAY entre sí, pues no tendría sentido, ni hacer operaciones con ellas tomadas en su conjunto. La única operación posible con todos los elementos de un ARRAY a la vez es la asignación:

```
NOTASCLASE1:=NOTASCLASE2;
```

esto guardaría todos los valores de la tabla NOTASCLASE2 en la tabla NOTASCLASE1 , es decir, el elemento 1 de la tabla NOTASCLASE2 en el 1 de NOTASCLASE1, el 2 en el 2, etc.

Hay que hacer notar que el número de elementos de la tabla queda definido por el subrango del índice y éste es fijo. Si al escribir el programa especificamos, por ejemplo, 21..30, la tabla tendrá 10 elementos y esto no podrá ser cambiado durante la ejecución del programa.

Si quisiéramos preparar el programa NOTAS para funcionar con una tabla en lugar de con una variable distinta para cada nota, pondríamos en principio

```
............;
WRITELN('INTRODUZCA LAS NOTAS');
READLN (NOTA[1]);
READLN (NOTA[2]);
READLN (NOTA[3]);
................;
```

con lo que ciertamente se gana poco. Sin embargo, como el índice puede ser una variable, el paquete de instrucciones READLN se puede sustituir por un bucle FOR en que la variable de control se utilice como índice, y lo mismo sucede para las demás instrucciones que se repiten.

Estamos ya en condiciones de hacer un programa de medias mejorado;

```
PROGRAM NOTASMEJOR;
CONST
  TOPE=100;  (* Como mucho podrá haber 100 notas *)
TYPE
  TIPOINDICE= 1..TOPE;
  TIPONOTAS = ARRAY [TIPOINDICE] OF REAL;
  (* Las notas pueden tener decimales, de ahí el tipo REAL *)
VAR
```

```
    NOTAS               : TIPONOTAS;
    SUMA, MEDIA, MAXIMA : REAL;
    INDICE,TOTAL        : TIPOINDICE;

  BEGIN
   WRITELN ('EL MAXIMO NUMERO DE NOTAS ES ',TOPE);
   WRITE ('CUANTAS NOTAS ? '); READLN(TOTAL);
  (* Vamos a leer las notas tecleadas *)
   FOR INDICE:=1 TO TOTAL DO
     BEGIN
      WRITE ('NOTA NUMERO ',INDICE,' = ');
      READLN (NOTAS[INDICE])
      (* Como un REAL se puede leer de teclado, los elementos de NOTAS
         también *)
     END;

  (* Ahora calculamos la media *)
   SUMA:=0.0;     (* Vamos acumulando notas en SUMA: *)
   FOR INDICE:=1 TO TOTAL DO SUMA:=SUMA+NOTAS[INDICE];
   MEDIA:=SUMA/TOTAL;
   WRITELN('La nota media es',MEDIA:6:2);

   (* Ahora obtenemos la nota máxima *)
   MAXIMA:=0.0;
   FOR INDICE:=1 TO TOTAL DO
     IF NOTAS[INDICE]>MAXIMA THEN MAXIMA (*por ahora*) :=NOTAS[INDICE];
   WRITELN('La nota más alta es:',MAXIMA:6:2);
  END.
```

Como se ve, en lugar de utilizar una instrucción para cada nota se han utilizado bucles FOR, pues se repiten un número fijo de veces. Además, se ha descrito el subrango del índice previamente para así no tenerlo que repetir con INDICE y TOTAL. Nótese también dónde se ha puesto uno de los comentarios; recordemos que se pueden poner comentarios en cualquier lugar en que pueda ir un espacio en blanco.

Hasta ahora sólo se han utilizado índices de tipo subrango del tipo INTEGER. Sin embargo, se pueden utilizar índices de cualquier tipo subrango o escalar siempre que la tabla resultante quepa junto con las demás variables en la memoria del ordenador.

Por ejemplo, para guardar los gastos de los diferentes días de la semana y las horas trabajadas en un programa de presupuestos podríamos poner:

```
TYPE
 DIADELASEMANA = (LUNES,MARTES,MIERCOLES,JUEVES,VIERNES,SABADO,DOMINGO);
 DIALABORABLE  = LUNES..VIERNES;
```

```
      TIPOGASTOS    = ARRAY [DIADELASEMANA] OF REAL;
      TIPOHORAS     = ARRAY [DIALABORABLE ] OF INTEGER;
    VAR
      GASTOS: TIPOGASTOS;
      HORAS : TIPOHCRAS;
      DIA   : DIADELASEMANA;
```

y sería posible escribir entonces cosas como:

```
HORAS [SUCC(MARTES)]:=5; (* O sea, el Miércoles 5 horas *)
FOR DIA:=LUNES TO DOMINGO DO WRITELN (GASTOS[DIA]:7:0);
```

Los elementos constitutivos de una variable de tipo ARRAY pueden ser de absolutamente cualquier tipo. Tipos estándar, tipos nuevos, incluso ARRAYS.

Imaginemos que en el colegio en que se va a utilizar el programa de medias hay varios grupos llamados A, B, C, D y E. Una vez más podríamos crear para cada uno una variable tipo ARRAY propia, NOTASA, NOTASB, etc., pero entonces habría que repetir las instrucciones para cada una.

En lugar de eso, podemos crear una tabla de ... tablas de notas:

```
TYPE
  TIPOALUMNO= 1..100;
  TIPOGRUPO = ARRAY [TIPOALUMNO] OF REAL;
  TIPONOTAS = ARRAY ['A'..'E'] OF TIPOGRUPO;
(*   O bien ARRAY ['A'..'E'] OF ARRAY [TIPOALUMNO] OF REAL   *)
VAR
  NOTAS : TIPONOTAS;
```

Para referirnos entonces a la tabla de notas del grupo B pondríamos:

```
NOTAS ['B']
```

Y para referirnos a la nota 37 del grupo B, por tanto:

```
NOTAS ['B'][37]
```

Teniendo en cuenta las diferentes asignaturas, podríamos tener:

```
TYPE
  ASIGNATURAS=(FISICA,LENGUA,INGLES);
  TIPOALUMNO= 1..100;
  TIPONOTAS= ARRAY [ASIGNATURAS] OF ARRAY ['A'..'E'] OF ARRAY
             [TIPOALUMNO] OF REAL;
VAR NOTAS: TIPONOTAS;
```

y tendríamos entonces, por ejemplo:

```
NOTAS [FISICA]['B'][37]
```

que sería la nota del alumno 37 del grupo B en Física.

A este tipo de variables se les llama ARRAYs MULTIDIMENSIONALES. Como se ve, el total de notas que se podría guardar en NOTAS sería 3 asignaturas por 5 grupos por 100 alumnos, igual a 1500.

En PASCAL las expresiones como ARRAY[...] OF ARRAY[...] OF ... y NOTAS[ ][ ][ ] se pueden abreviar así:

```
TIPONOTAS = ARRAY [ASIGNATURAS, 'A'..'E', TIPOALUMNO]
            OF REAL
y
NOTAS [FISICA, 'B',37]
```

Array de caracteres

Un caso especial de ARRAY es aquél cuyos elementos son caracteres. Al ser tablas de éstos, permiten guardar textos.

Supongamos que tenemos definida la variable TEXTO como ARRAY[1..4] OF CHAR ; como hemos visto, para guardar la palabra 'PEPE' habría que hacer

```
TEXTO[1]:='P'; TEXTO[2]:='E'; TEXTO[3]:='P'; TEXTO[4]:='E';
```

Sin embargo, la mayoría de los compiladores permiten poner

```
TEXTO:='PEPE'; (* Deben ser exactamente 4 caracteres *)
```

También suele ser posible escribir o leer con una sola instrucción estas variables:

```
READLN (TEXTO);
WRITELN (TEXTO);
```

Esta última equivaldría a

```
FOR INDICE:=1 TO 4 DO WRITE (TEXTO[INDICE]); WRITELN;
```

Por otra parte, la instrucción de lectura tomaría sólo cuatro caracteres de los tecleados si hubiera más, y rellenaría con espacios en blanco (en algunos casos con el carácter cuyo ordinal es 0, CHR(0)) los que faltaran hasta cuatro si se hubieran tecleado menos antes de pulsar RETURN, INTRO, etcétera.

En aquellos casos en que existe el tipo predefinido STRING, una variable de este tipo normalmente puede ser asignada a un ARRAY OF CHAR si la longitud del string en ese momento coincide con la del ARRAY, y a la inversa en todo caso.



Ejemplo

Para terminar y a modo de ejemplo vamos a ver cómo podríamos guardar en memoria la situación del tablero contrario en una partida de batalla naval (o de barquitos).

En cada punto del tablero puede haber cuatro situaciones: que haya agua, que haya un barco tocado o hundido o que no se sepa lo que hay. Podríamos así definir el tipo de los posibles valores que puede tomar un punto:

```
TYPE
 POSIBLE=(AGUA,BARCOTOCADO,BARCOHUNDIDO,DESCONOCIDO);
```

Una fila del tablero sería entonces un ARRAY [1..10] OF POSIBLE y, por tanto, el tablero completo quedaría como un ARRAY de filas, o sea:

```
VAR
  CONTRARIO: ARRAY['A'..'J'] OF ARRAY[1..10] OF POSIBLE;
```

o bien

```
CONTRARIO: ARRAY ['A'..'J', 1..10] OF POSIBLE;
```

entonces podríamos escribir cosas como

```
CONTRARIO ['B',7]:=AGUA; (* O sea, en B7 hay agua *)
IF CONTRARIO['J',9] = DESCONOCIDO THEN ...
```

Si quisiéramos visualizar en pantalla la situación del tablero escribiendo, por ejemplo, un espacio en blanco en las casillas desconocidas, una T en donde haya barco tocado, una H en los hundidos y una A donde se sepa que hay agua, podríamos escribir:

```
CLRSCR; (* o PAGE o como se deba hacer para borrar la pantalla *)

FOR FILA:='A' TO 'J' DO            (* Dibujar de fila en fila: *)
 BEGIN
  (* Primero pintar la fila de columna en columna: *)
  FOR COLUMNA:=1 TO 10 DO
   BEGIN
    IF CONTRARIO [FILA,COLUMNA] = DESCONOCIDO  THEN WRITE (' ');
    IF CONTRARIO [FILA,COLUMNA] = BARCOTOCADO  THEN WRITE ('T');
    IF CONTRARIO [FILA,COLUMNA] = BARCOHUNDIDO THEN WRITE ('H');
    IF CONTRARIO [FILA,COLUMNA] = AGUA         THEN WRITE ('A')
   END;
  WRITELN   (* Tras acabar cada fila, bajar una línea *)
 END;
```

FILA y COLUMNA deberían ser variables de tipo CHAR e INTEGER, respectivamente (o subrango de ellos).

El PASCAL permite así definir estructuras de datos sumamente adaptadas al problema real que se desee resolver. Esto redundará en una mayor claridad de los programas y, por tanto, en una programación más eficiente y depurada con menor cantidad de errores.

PROCEDIMIENTOS Y FUNCIONES 8



UANDO escribimos el programa de descomposición en factores lo hicimos siguiendo una técnica de refinamiento gradual, es decir, descomponiendo primero el problema en sus partes principales y analizando luego cada una de estas partes por separado para descomponerlas a su vez en tareas más sencillas. Así, teníamos un primer esquema del programa como éste:

1. Leer primer número.
2. Mientras (while) el número sea distinto de cero hacer:
   - Procesar y sacar resultados.
   - Leer siguiente número.
3. Mostrar en pantalla mensaje de despedida.

A su vez, «procesar y sacar» se descomponía en varios pasos:

A. Anotar que NUMERO es primo.
B. Para FACTOR valiendo desde 2 hasta NUMERO DIV 2 hacer:
   Si NUMERO dividido por FACTOR es un número entero entonces:
   - Presentar FACTOR y NUMERO DIV FACTOR.
   - Tachar la anotación de qué NUMERO es primo.
C. Si todavía está la nota, avisar de que NUMERO es primo.

Como culminación de este proceso, hubo que juntar todos los pasos sencillos en que llegamos a descomponer el problema para construir el programa definitivo.

Si, teniendo ya el programa terminado, se descubren errores o se desea modificar alguna de las partes integrantes del programa, hay dos opciones; o bien se vuelve a repetir todo el proceso de descomposición y posterior integración, o se introducen los cambios directamente en el progra-

ma ya escrito. En el primer caso, si el programa es grande supondría desperdiciar una gran cantidad de trabajo ya hecho.

En el segundo caso, dado que ya están todas las partes integradas, habría que delimitar claramente el área afectada para introducir los cambios y comprobar que éstos no afectan a las otras partes.

Esta tarea sería mucho más sencilla si el programa PASCAL reflejara los diferentes niveles de descomposición del problema. Así, la parte principal del programa se correspondería más o menos con el primer nivel de descomposición, indicando las diferentes fases del proceso. Luego, cada una de éstas estaría programada por separado constituyendo lo que denominaríamos subprogramas.

Al ejecutarse el programa, la parte principal iría utilizando a los diferentes subprogramas según fuera preciso.

Con ello, el punto 2 del programa de los factores se escribiría:

```
WHILE (NUMERO<>0) DO
  BEGIN
    PROCESARYMOSTRAR;
    WRITELN ('Introduzca número a descomponer.');
    READLN (NUMERO); NUMERO:=ABS(NUMERO)
  END;
```

PROCESARYMOSTRAR sería el subprograma donde ya sí estaría detallado en qué consiste «procesar y sacar». En caso de querer hacer algún cambio en la parte del proceso, no habría que modificar nada del programa principal; todos los cambios serían únicamente en el subprograma.

A su vez, el subprograma podría hacer referencia de manera análoga a otros subprogramas de nivel inferior que realizaran alguna de sus partes constitutivas.

Esto es posible en PASCAL mediante el uso de PROCEDIMIENTOS y FUNCIONES, que son conjuntos de instrucciones a los que se les ha dado un nombre o identificador. Para ser utilizados, basta con escribir éste como una instrucción más, o sea, separado de las demás por punto y coma.

Cuando se ejecuta el programa, al llegar a un nombre de procedimiento o función se pasa a ejecutar su conjunto de instrucciones. Tras ellas se siguen ejecutando las que vengan a continuación del nombre.

La única diferencia entre procedimientos y funciones es que estas últimas además devuelven un valor (un número, un carácter...) para su uso por el programa principal.

En PASCAL existen ya procedimientos y funciones previamente programados. Un ejemplo de procedimiento es CLRSCR (o PAGE, etc.). Cuando ponemos:

```
CLRSCR;
WRITE('Se acaba de borrar la pantalla');
```

al ejecutarse CLRSCR, realmente se ejecuta un conjunto de instrucciones con ese nombre. Tras ello, se ejecutaría la siguiente instrucción, WRITE.

Un ejemplo de función es ODD. Está formada por un conjunto de instrucciones que, al estar ya preparadas, no tienen que escribirse. A estas instrucciones se les suministra un dato de tipo INTEGER y, según que sea impar o no, devuelven el valor de tipo Boolean TRUE o FALSE, respectivamente. El dato a analizar se escribe a continuación del nombre de la función, entre paréntesis, y el dato Boolean devuelto pasa a «ocupar», por decirlo de alguna manera, el sitio del nombre de la función. Por ejemplo:

```
N:=2;
A:=ODD(N+1);
```

Al llegar al nombre ODD se pasa a ejecutar su conjunto de instrucciones, a las que se transfiere el número 3. Estas lo analizan y devuelven TRUE, por lo que en ese momento esa instrucción equivaldría a A:=TRUE.

Ejemplos de procedimientos a los que se transfieren datos para procesar son WRITE y READ.

Funciones ya estudiadas son PRED, SUCC, ORD, SIN, COS, etc.

Otra ventaja de los procedimientos y funciones, casi tan importante como la de poder estructurar los programas, es la de permitir hacer los programas más cortos.

En efecto, supongamos que hemos preparado una secuencia de instrucciones para, por ejemplo, mostrar por pantalla la situación del tablero de barquitos del capítulo anterior. Si al escribir el programa resultara que hay que mostrar tableros en diferentes momentos de su ejecución, no habría más remedio que repetir esas instrucciones en todos los puntos necesarios.

En lugar de eso, podríamos agrupar esas instrucciones como un procedimiento de nombre, digamos, PRESENTAR. Entonces bastaría con escribir PRESENTAR en todos los puntos del programa necesarios.

Si además hubiera que mostrar varios tableros distintos, tendríamos que transferir al procedimiento el nombre de la variable en cuestión. Así, si escribiéramos en el programa principal:

```
PRESENTAR (CONTRARIO1);
PRESENTAR (CONTRARIO2);
```

mostraría primero la situación del tablero guardada en la variable CONTRARIO1 y luego la de CONTRARIO2.

Vamos a estudiar a continuación cómo utilizar los procedimientos y las funciones.



ESCRIBIENDO PROCEDIMIENTOS

Los procedimientos se escriben después de la zona de descripción de datos del programa, justo antes de la palabra reservada BEGIN, que marca el comienzo de la zona de instrucciones del programa.

De esta manera, la estructura de un programa PASCAL quedaría:

PROGRAM NombreDelPrograma;

Descripción de datos:

CONST ...

TYPE ...

VAR ...

Procedimientos y funciones.

BEGIN

Instrucciones del programa.

END.

De las diferentes partes del programa, sólo la cabecera y las palabras BEGIN y END con punto son obligatorias en todo programa PASCAL. Las diferentes partes van separadas entre sí por punto y coma.

Supongamos que queremos escribir un programa que cuente del 1 al 10. Tendría la siguiente estructura:

1. Presentar mensaje en pantalla indicando lo que hace.
2. Para CUENTA valiendo desde 1 hasta 10 hacer:
   — Presentar CUENTA en pantalla.
   — Pasar a siguiente línea.
3. Presentar mensaje de despedida.

A modo de ejemplo, vamos a escribir los puntos 1 y 3 como procedimientos. Tendríamos un programa como:

```
PROGRAM CONTAR;
 VAR CUENTA: INTEGER;
BEGIN
 PROLOGO;
 FOR CUENTA:=1 TO 10 DO WRITELN (CUENTA:3);
 DESPEDIDA
END.
```

Al intentar compilar el programa se producirían errores, pues al buscar el compilador dónde están descritos los procedimientos no los encontraría.

La descripción de un procedimiento comienza por la palabra reservada PROCEDURE seguida de su nombre, que puede ser cualquier identificador válido. Esto es lo que se denomina cabecera del procedimiento, que debe ir separada de lo siguiente por un punto y coma.

Además, todo procedimiento consta de las palabras reservadas BEGIN y END, que marcan el principio y final de sus instrucciones.

Las descripciones de los diferentes procedimientos y funciones se escriben unas detrás de otras, separándose la cabecera de uno de la palabra END del anterior mediante un punto y coma.

Por tanto, para que el compilador no produjera errores bastaría con poner:

```
PROGRAM CONTAR;
 VAR CUENTA: INTEGER;

PROCEDURE PROLOGO;
```

```
          BEGIN
          END;
         PROCEDURE DESPEDIDA;
          BEGIN
          END;

         BEGIN
          PROLOGO;
          FOR CUENTA:=1 TO 10 DO WRITELN (CUENTA:3);
          DESPEDIDA
         END.
```

Al ejecutarse este programa, como las descripciones de los procedimientos no constan de ninguna instrucción, es como si hubiéramos puesto «no hacer nada», con lo que sólo saldrá la cuenta en la pantalla. Nuestra intención es que los procedimientos saquen unos mensajes y, por tanto, deben tener escritas las instrucciones necesarias para ello, con lo que el programa definitivo quedaría así:

```
PROGRAM CONTAR;
 VAR CUENTA: INTEGER;

PROCEDURE PROLOGO;
 BEGIN
  CLRSCR;   (* o PAGE o como se haga para borrar la pantalla. *)
  WRITELN('Este programa sólo cuenta desde 1 hasta 10.');
  WRITELN('Vean si no:')
 END;

PROCEDURE DESPEDIDA;
 BEGIN
  WRITELN('Tras esto, no hay más. Adiós.')
 END;

BEGIN
 PROLOGO;
 FOR CUENTA:=1 TO 10 DO WRITELN (CUENTA:3);
 DESPEDIDA
END.
```

Por supuesto, las instrucciones utilizadas en los procedimientos van separadas entre sí por punto y coma, y pueden ser de cualquiera de los tipos estudiados.



VARIABLES LOCALES

A los procedimientos y funciones se les conoce en general como subprogramas, o sea, programas de orden menor para ser utilizados por otros de orden superior.

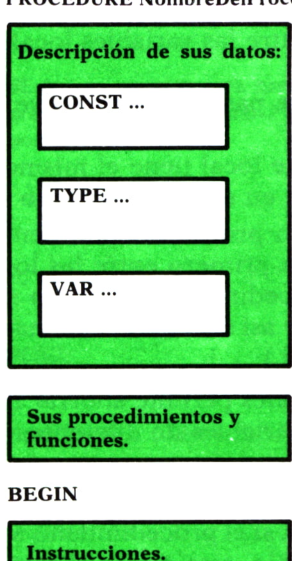
Si recordamos los primeros capítulos, veremos que la estructura de los procedimientos del ejemplo es similar a la de aquellos programas. En ambos casos existe una cabecera que comienza por las palabras reservadas PROCEDURE o PROGRAM, según el caso.

Tras la cabecera viene el conjunto de instrucciones enmarcado por las palabras BEGIN y END, habiendo un punto a continuación de END en el caso de los programas para indicar el final absoluto del programa y un punto y coma en los procedimientos para separarlos de lo que venga a continuación.

Pues bien, el paralelismo va mucho más lejos.

Los procedimientos pueden tener su propia zona de definición de datos con variables para su uso exclusivo. Incluso pueden tener a su vez sus propios procedimientos y funciones.

La estructura quedaría, por tanto, así:



Es decir, son como programas **PASCAL** escritos dentro del programa principal.

Tanto las constantes como los tipos de datos, variables, procedimientos y funciones definidos dentro de un procedimiento se denominan **LOCALES** de ese procedimiento y son para su uso exclusivo. Solamente pueden ser utilizados dentro de él (y de los propios procedimientos y funciones de éste).

Sin embargo, todas las constantes, tipos y variables del programa principal siguen pudiendo ser utilizados por las instrucciones del procedimiento.

Las variables locales de un procedimiento son iguales en casi todos los aspectos a las del programa principal, denominadas a su vez **GLOBALES** por poder utilizarse en cualquier parte del programa, tanto en la parte principal como en cualquier procedimiento o función.

La única diferencia estriba en que sólo toman la porción de memoria que necesitan en el momento en que se utiliza el procedimiento. Cuando se termina de ejecutar éste, el espacio de memoria ocupado por sus variables queda libre para su uso en cualquier otra cosa, como por ejemplo, para las variables de otro procedimiento que se ejecute a continuación.

Por ello, no es posible conservar datos en una variable local entre dos utilizaciones de un procedimiento. Una vez acabado éste, se pierden todos sus datos locales. Si se necesitara guardar algún dato entre ejecuciones sucesivas habría que utilizar variables globales.

Cuando el **PASCAL** encuentra el nombre de una variable busca primero a ver si la encuentra entre las variables locales del procedimiento y es ésa la que utiliza.

Sólo si no la encuentra pasa a buscar entre las variables globales del programa.

Por ello, si una variable local tiene el mismo nombre que una global, la que se utilizará siempre en el procedimiento será la variable local.

(Cuando se trata de un procedimiento, dentro de un procedimiento, dentro de otro, etc., busca primero entre las locales propias; si no se encuentra, entre las del procedimiento de orden superior, que a su vez son globales para el anterior, y así sucesivamente hasta acabar buscando entre las del programa principal.)

El uso de variables locales permite ahorrar memoria, pues las mismas zonas son utilizadas por variables de diferentes procedimientos, según el momento.

Sin embargo, las principales ventajas son otras.

Por un lado, evitan que, si el procedimiento altera (erróneamente o no) el contenido de una variable, esto afecte al resto del programa, cosa que sucedería si la variable fuera global.

Por otro lado, permiten escribir procedimientos autosuficientes, es de-

cir, que no necesitan que se les prepare en la zona de definición de datos del programa ninguna variable especial para ellos.

Así, procedimientos escritos y probados en alguna ocasión para otros programas pueden ser utilizados sin más que copiar su descripción. De esta manera se ahorra una enorme cantidad de trabajo y se evitan posibles errores.

Vamos a ver a continuación un sencillo ejemplo de utilización de variables locales. Supongamos que ahora deseamos que el programa CONTAR tenga también el punto 2 escrito como procedimiento y que éste utilice sólo variables locales; además, para ilustrar los casos de coincidencia de nombre entre variables globales y locales, vamos a hacer que la cuenta desde 1 hasta 10 se repita cinco veces, utilizando una variable global, CUENTA, para ello. El punto 2 sería ahora algo como:

2. Para CUENTA valiendo desde 1 hasta 5 hacer:

   — Contar desde 1 hasta 10.
   — Avisar del final de la cuenta.

y «Contar desde 1 hasta 10» sería el antiguo punto 2 que pondremos ahora como procedimiento. El programa quedaría:

```
PROGRAM CONTAR2;
 VAR CUENTA: INTEGER; (* Esta variable es global *)

PROCEDURE PROLOGO;
 BEGIN
  CLRSCR;   (* o PAGE o como se haga para borrar la pantalla. *)
  WRITELN('Este programa sólo cuenta desde 1 hasta 10.');
  WRITELN('Vean si no:')
 END;

PROCEDURE DESPEDIDA;
 BEGIN
  WRITELN('Tras esto, no hay más. Adiós.')
 END;

PROCEDURE CONTARUNOADIEZ;
 VAR CUENTA: INTEGER; (* Esta variable es local de CONTARUNOADIEZ *)
BEGIN
 FOR CUENTA:=1 TO 10 DO WRITELN (CUENTA:3);
END;

BEGIN
 PROLOGO;
 FOR CUENTA:=1 TO 5 DO   (* Esta variable es la global *)
   BEGIN
```

```
        CONTARUNOADIEZ;
        WRITELN ('Fin de la cuenta.')
       END;
     DESPEDIDA
    END.
```

Desde luego, utilizar nombres coincidentes sólo sirve para hacer los programas menos claros y, por ello, sólo es aconsejable para variables que se usen muchas veces para diferentes tareas sencillas, sin una misión específica (a estas variables es una práctica usual darles nombres cortos como **I**, **J**, **N**, **R** y son las típicas a usar, por ejemplo, para contar el número de veces que se repite un bucle, para una entrada de datos en la que se quiere comprobar el rango antes de utilizar la variable definitiva, etc.).



PASO DE PARAMETROS POR VALOR

Si se necesitara que un procedimiento manejase datos pertenecientes al programa principal (o al procedimiento en que estuviera inserto, caso de que fuese de menor nivel) una posible forma sería utilizar la o las variables globales en que estuvieran guardados esos datos llamándolas por su nombre y cuidando, por supuesto, de que no hubiera variables locales con igual denominación.

Partiendo del programa CONTAR2, vamos a hacer otro que realice una sola cuenta desde 1 hasta un valor introducido por teclado en la variable global TOPE:

```
PROGRAM CONTAR3;
 VAR TOPE: INTEGER;

PROCEDURE CONTARUNOA;
(* Necesita el dato TOPE del programa principal *)
 VAR CUENTA: INTEGER;
BEGIN
 FOR CUENTA:=1 TO TOPE DO WRITELN (CUENTA:3);
END;
BEGIN
 (* Por esta vez, pasamos de prólogo *)
 WRITE ('Contar del uno al...');
 READLN (TOPE);
 CONTARUNOA;
 (* Pasamos de despedida *)
END.
```

Sin embargo, este método obliga a que el programa principal y el procedimiento se pongan de acuerdo en la variable a utilizar, como sucede con TOPE, y ésta tendrá que llamarse igual dentro y fuera del procedimiento. Así, la «autosuficiencia» o facilidad de exportar procedimientos de unos programas a otros resulta bastante menor. Además, no podríamos suministrar al procedimiento los resultados de expresiones ni los datos contenidos en otras variables diferentes a las convenidas sin guardarlos previamente en éstas cada vez que hubiera que ejecutarlo.

Por otra parte, tampoco se pueden utilizar variables locales normales para pasar los datos, pues ya se ha dicho que el programa principal no tiene acceso a ellas, al existir éstas en memoria sólo durante la ejecución del procedimiento, por lo que no sería posible guardar los datos en ellas justo antes de ejecutarlo.

Para solucionar esto, existe en PASCAL la posibilidad de transferir datos al tiempo que se llama al procedimiento por medio de ciertas variables locales especiales.

Para ello, en su cabecera y a continuación del nombre se deben definir las variables locales en las que se van a guardar los datos a transferir en el momento de la llamada, describiendo lo que se denomina lista de parámetros.

Esto se hace de manera idéntica a como se definen las variables en la zona de descripción de datos, es decir, el nombre de cada variable debe ir seguido de dos puntos y de su tipo, tras lo cual podrían venir las definiciones de otras variables separadas entre sí por punto y coma. En caso de haber varias variables del mismo tipo, se podrían agrupar separadas por comas, tras lo cual vendrían los dos puntos y el tipo común. Toda la lista de parámetros debe ir entre paréntesis. Cabeceras válidas serían, por tanto:

```
PROCEDURE AAAA ( ENE: INTEGER );

PROCEDURE BBBB ( MAX: INTEGER; R: REAL );

PROCEDURE CCCC ( I,J,K: INTEGER; D: DIADELASEMANA; B: BOOLEAN );

PROCEDURE DDDD ( H: DIALABORABLE; M: DIALABORABLE);
```

Las variables descritas en la lista de parámetros son unas variables locales más, con la única particularidad de que se guardan datos en ellas justo en el momento de llamar al procedimiento.

Los datos a guardar se deben indicar entre paréntesis y separados entre sí por comas, justo a continuación del nombre del procedimiento, cada vez que se utilice éste. Deben ir exactamente en el mismo orden en que están definidas las variables en la cabecera.

En la lista de parámetros a transferir pueden aparecer tanto constantes como variables o expresiones siempre que el tipo coincida con el indicado en la cabecera. Así

```
AAAA (17)
```

haría que ENE valiera 17 nada más comenzarse a ejecutar AAAA; con

```
BBBB ( 5 + 4, 6.0 / 3.0 )
```

MAX valdría 9 y R 2.0.

```
CCCC (TOPE,CUENTA,3, SUCC(LUNES), 2>3)
```

sería también correcto y con ello I y J tomarían los valores de TOPE y CUENTA en el momento de la llamada y K,D y B valdrían 3, MARTES y FALSE respectivamente. Sin embargo,

```
DDDD ( LUNES ) y AAAA ( FALSE )
```

producirían errores en la compilación pues DDDD precisa dos parámetros y el tipo del parámetro de AAAA es INTEGER.

Revisemos ahora el programa CONTAR3. Para ello vamos a transferir como parámetro el valor final de la cuenta:

```
PROGRAM CONTAR3MEJOR;
 VAR TOPE: INTEGER;

PROCEDURE CONTARUNOA (FINAL: INTEGER);
(* Cuenta desde 1 hasta FINAL escribiendo en pantalla. *)
 VAR CUENTA: INTEGER;
BEGIN
 FOR CUENTA:=1 TO FINAL DO WRITELN (CUENTA:3);
END;
```

```
BEGIN
 WRITE ('Contar del uno al...');
 READLN (TOPE);
 CONTARUNOA (TOPE);
END.
```

Aquí se observa una de las ventajas más importantes de los procedimientos:

Si el procedimiento CONTARUNOA hubiera sido escrito por otro programador, nosotros no necesitaríamos conocer absolutamente nada de sus interioridades para utilizarlo. Bastaría con copiarlo y saber, primero, que se llama CONTARUNOA y, segundo, que precisa de un único parámetro de tipo INTEGER que indica el final de la cuenta. Nada más. Por ello es una buena costumbre describir con un comentario para qué sirve un procedimiento justo a continuación de su cabecera.

Como se mencionó en su momento, WRITE y WRITELN no se pueden utilizar con tipos no estándar. Estamos ya en condiciones de preparar nuestro propio procedimiento de escritura para variables del tipo DIADELASEMANA:

```
PROCEDURE WRITEDIA (D: DIADELASEMANA );
 BEGIN
  IF D=LUNES     THEN WRITE('Lunes')     ELSE
  IF D=MARTES    THEN WRITE('Martes')    ELSE
  IF D=MIERCOLES THEN WRITE('Miércoles') ELSE
  IF D=JUEVES    THEN WRITE('Jueves')    ELSE
  IF D=VIERNES   THEN WRITE('Viernes')   ELSE
  IF D=SABADO    THEN WRITE('Sábado')    ELSE
                      WRITE('Domingo')
 END;
```

Para presentar el día guardado en la variable HOY bastaría con escribir

```
WRITEDIA (HOY);
```

siempre que hiciera falta.

Realmente, WRITE, WRITELN, READ y READLN son unos procedimientos algo especiales, pues se les puede transferir un número variable

de parámetros, cosa que no se puede hacer con los programados por nosotros.



PASO DE PARAMETROS POR NOMBRE

Supongamos ahora que queremos escribir además el procedimiento **READLNDIA** de manera que, por ejemplo, guarde **LUNES**, **MARTES**, etc., si se teclea L,M,X,J,V,S y D. En principio podría ser algo como:

```
PROCEDURE READLNDIA (D: DIADELASEMANA );
 VAR C: CHAR;
BEGIN
 READLN (C);
 IF C='L' THEN  D:=LUNES     ELSE
 IF C='M' THEN  D:=MARTES    ELSE
 IF C='X' THEN  D:=MIERCOLES ELSE
 IF C='J' THEN  D:=JUEVES    ELSE
 IF C='V' THEN  D:=VIERNES   ELSE
 IF C='S' THEN  D:=SABADO    ELSE
 IF C='D' THEN  D:=DOMINGO
END;
```

Entonces, para leer la variable **DIA** haríamos:

```
READLNDIA (DIA);
```

Sin embargo, debemos recordar que la variable **D** descrita en la cabecera es local y, por tanto, por mucho que cambiemos su valor, **DIA** permanecerá invariable.

Podríamos escribir **READLNDIA** de manera que él mismo guardara el dato en la variable **DIA** poniendo, por ejemplo, **DIA:=D** como última instrucción, pero está claro que entonces sólo serviría para esa variable.

Para solucionar estas situaciones está lo que se denomina *transferencia de parámetros por nombre*.

Utilizando ésta modalidad, las variables descritas en la cabecera no son locales del procedimiento y cada vez que se ejecuta éste, la porción de memoria que les corresponde es precisamente la de las variables que aparecían en la lista de parámetros en el momento de la llamada.

En otras palabras, la variable de la cabecera es en cada llamada exac-

tamente la misma que figura en la lista de parámetros, de manera que si modificamos su valor modificamos también el de esta última.

Para indicar que un parámetro se transfiere por nombre se debe poner delante de su nombre la palabra reservada VAR:

```
PROCEDURE PAF (I: INTEGER; VAR J,K: INTEGER; B: BOOLEAN; VAR C: CHAR);
```

en este procedimiento, J,K y C se transfieren por nombre y los demás por el método habitual.

Evidentemente, cuando un dato se transfiere por nombre, sólo puede aparecer una variable en la lista de parámetros; no están permitidas ni constantes ni expresiones.

Veamos un ejemplo:

```
PROGRAM EJEMPLOPORNOMBRE;
 VAR NUMERO: INTEGER;

 PROCEDURE PONERACERO (VAR N: INTEGER);
 (* Pone a cero la variable transferida *)
 BEGIN
  WRITELN ('La variable transferida valía ',N);
  WRITELN ('Pero desde ahora valdrá 0');
  N:=0
 END;

BEGIN
 NUMERO:=7; (* NUMERO ahora vale 7 *)
 WRITELN ('Número=',NUMERO);
 PONERACERO (NUMERO);  (* Pero ahora vale 0 *)
 WRITELN ('Número=',NUMERO)
END.
```

Si quitáramos ahora la palabra VAR de la cabecera de PONERACERO, comprobaríamos que NUMERO no cambia de valor.

Estamos ya en condiciones de escribir el procedimiento READLNDIA:

```
PROCEDURE READLNDIA (VAR D: DIADELASEMANA );
  VAR C: CHAR;
 BEGIN
  READLN (C);
```

```
IF C='L' THEN D:=LUNES     ELSE
IF C='M' THEN D:=MARTES    ELSE
IF C='X' THEN D:=MIERCOLES ELSE
IF C='J' THEN D:=JUEVES    ELSE
IF C='V' THEN D:=VIERNES   ELSE
IF C='S' THEN D:=SABADO    ELSE
IF C='D' THEN D:=DOMINGO
END;
```

Hay una ventaja adicional en el uso del paso de parámetros por nombre, que debe utilizarse con sumo cuidado.

Cuando el dato a transferir ocupa mucha memoria (por ejemplo, una variable ARRAY multidimensional de, digamos, treinta mil números enteros), al utilizar el procedimiento por el método habitual se crea una variable local de las mismas dimensiones con lo que las necesidades de memoria son muy grandes. Además, cada vez que se utiliza se produce una transferencia automática de los datos a la variable local y como son muchos el programa puede llegar a hacerse más lento.

Pasando los datos por nombre, la porción de memoria que se utiliza es la misma del original y, por tanto, no se necesita tanta memoria ni emplear tanto tiempo en copiar los datos. Eso sí, existe el riesgo de que por una mala programación alguno de los datos originales resulte alterado de manera involuntaria al ejecutarse el procedimiento.



FUNCIONES

Las funciones son un tipo especial de procedimientos que, además de permitir hacer todo lo que hemos visto hasta ahora, devuelven un valor que, para entendernos, se podría decir que pasa a ocupar el lugar del nombre de la función en el punto en que se llamó a ésta. Este valor puede ser de cualquier tipo no estructurado, es decir, valores simples como INTEGER, REAL, CHAR, BOOLEAN o tipos escalares de nueva creación.

Ya conocemos ejemplos de funciones predefinidas como son las de tipo REAL COS y SIN y la de tipo BOOLEAN ODD. Para utilizarlas no se necesita conocer nada sobre cómo han sido programadas. Además, se pueden utilizar como parámetros con ellas tanto variables como expresiones o constantes.

En esta misma línea, es una buena práctica escribir las funciones de manera que se puedan utilizar de manera similar a las anteriores, es decir, absteniéndose de utilizar en ellas variables globales y no utilizando el paso de

parámetros por nombre, de manera que la única consecuencia de su utilización esté en el resultado devuelto.

La única diferencia entre los procedimientos normales y las funciones a la hora de escribirlos está en la cabecera. En las funciones se utiliza la palabra reservada FUNCTION en lugar de PROCEDURE; además, a continuación del nombre de la función (o a continuación de la definición de la lista de parámetros, caso de existir éstos) se debe indicar el tipo de resultado que devuelve.

Por otra parte, cuando ya se haya obtenido el resultado, éste debe asignarse al nombre de la función como si de una variable se tratara.

Como ejemplo, vamos a escribir un programa que presente en pantalla el cubo de los diez primeros números naturales. Para ello utilizaremos la función CUBO, cuyo parámetro es INTEGER y que devuelve un resultado del mismo tipo:

```
PROGRAM LISTACUBOS;
 VAR ENE: INTEGER;

FUNCTION CUBO (X: INTEGER): INTEGER;
(* Devuelve el cubo de X *)
 BEGIN
  CUBO:=X*X*X
 END;

BEGIN                       (* Aquí comienza el programa principal *)
 FOR ENE:=1 TO 10 DO
  WRITELN ('El cubo de ',ENE:3,' es ',CUBO(ENE):6)
END.
```

La función CUBO así escrita es directamente utilizable por otros programas sin más que copiar la parte del programa PASCAL que le corresponde. Formas correctas de utilizarla serían:

```
A:= CUBO (2) - 3;              (* Guarda 5 en A *)
B:= CUBO (A-1);                (* Guarda 4 al cubo en B *)
WRITELN (CUBO(CUBO(3)-17));    (* Presenta 1000 en pantalla *)
```

Veamos otro ejemplo. Supongamos que la función ODD no existiera y que hubiera que escribirla. El parámetro es de tipo INTEGER y se devuelve el valor de tipo BOOLEAN TRUE o FALSE, según que aquél sea o no impar.

Para detectar si un número es impar, una posible manera es comprobar si el resto de dividirlo por dos es cero o uno:

```
FUNCTION ODD (N: INTEGER): BOOLEAN;
 BEGIN
  ODD:= ( (N MOD 2) = 1 )    (* Si N MOD 2 vale 1, ODD es TRUE *)
 END;
```

Esta función se utilizaría exactamente igual que la original.

Un último ejemplo.

En el procedimiento **READLNDIA**, que escribimos anteriormente, podría suceder que la letra tecleada fuera minúscula, con lo que en las instrucciones IF habría que poner

```
IF (C='L') OR (C='l') THEN D:=LUNES etc.
```

para permitir tanto mayúsculas como minúsculas.

En lugar de ello, vamos a utilizar la función **MAYUSCULA**, cuyo parámetro es un carácter y que devuelve el mismo carácter excepto en el caso en que aquél sea una letra minúscula, en que devuelve la mayúscula equivalente.

```
FUNCTION  MAYUSCULA(C: CHAR): CHAR;
 BEGIN
  IF ( 'a' <= C ) AND ( C <= 'z' )       (* Si C está entre a y z *)
    THEN
      MAYUSCULA:= CHR(ORD(C)-ORD('a')+ORD('A'))
    ELSE
      MAYUSCULA:=C  (* Se devuelve lo mismo en los demás casos *)
 END;
```

La operación ORD(C)-ORD('a') proporciona 0,1,2... según que C sea 'a','b','c'.... Si a este número le sumamos ORD('A'), obtendremos el número de orden de 'A','B','C'..., con lo que utilizando la función CHR se consigue la letra mayúscula equivalente. Esto, claro está, siempre que la letra sea una de las 26 del alfabeto inglés. Si además se necesitara la letra Ñ:

```
FUNCTION  MAYUSCULA(C: CHAR): CHAR;
 BEGIN
```

```
    IF ( 'a' <= C ) AND ( C <= 'z' )      (* Si C está entre a y z *)
      THEN
        MAYUSCULA:= CHR(ORD(C)-ORD('a')+ORD('A'))
      ELSE
        IF C='ñ' THEN MAYUSCULA:='Ñ' ELSE MAYUSCULA:=C
  END;
```

Esta función podría incorporarse a su vez como una función del procedimiento **READLNDIA** en el sitio que le corresponde, es decir, tras su zona de definición de datos y antes de sus instrucciones:

```
PROCEDURE READLNDIA (VAR D: DIADELASEMANA );
  VAR C: CHAR;


  FUNCTION  MAYUSCULA(C: CHAR): CHAR;
   BEGIN
    IF ( 'a' <= C ) AND ( C <= 'z' )   (* Si C está entre a y z *)
      THEN
        MAYUSCULA:= CHR(ORD(C)-ORD('a')+ORD('A'))
      ELSE
        MAYUSCULA:=C
   END;


BEGIN
 READLN (C);
 C:= MAYUSCULA (C);  (* Hacemos a C mayúscula *)
 IF C='L' THEN  D:=LUNES     ELSE
 IF C='M' THEN  D:=MARTES    ELSE
 IF C='X' THEN  D:=MIERCOLES ELSE
 IF C='J' THEN  D:=JUEVES    ELSE
 IF C='V' THEN  D:=VIERNES   ELSE
 IF C='S' THEN  D:=SABADO    ELSE
 IF C='D' THEN  D:=DOMINGO
END;
```

Por casualidad, tanto la variable de **READLNDIA** como el parámetro de **MAYUSCULA** tienen el mismo nombre; podrían llamarse de distinta manera, pero da lo mismo. La variable de **READLNDIA** es global para **MAYUSCULA**, pero como se empieza a buscar primero entre las locales propias, la que se utiliza en la función es la correcta.

No obstante, si con ello se evitan confusiones, no hay que dudar en cambiar los nombres.



RELACIONES ENTRE PROCEDIMIENTOS

La función MAYUSCULA inserta en READLNDIA sólo puede utilizarse desde las instrucciones de éste. Para el programa principal en que estuviera a su vez inserto el procedimiento, la función es un detalle de las interioridades de READLNDIA y como tal no tiene conocimiento de su existencia.

Podríamos haber escrito MAYUSCULA fuera de READLNDIA como una función independiente para poderla usar desde el programa principal, pero entonces se plantea la siguiente cuestión: ¿va a poder seguir siendo utilizada por el procedimiento?

Supongamos que tuviésemos un programa que se pudiera descomponer en tres partes:

— Fase A.
— Fase B.
— Fase C.

Si las fases A y B se descompusieran a su vez en, por ejemplo, las fases A1 y A2, B1 y B2, respectivamente, el esqueleto del programa PASCAL podría ser algo como:

```
PROGRAM RELACIONES;
(*---------------------------------------*)
 PROCEDURE FASEA;

   PROCEDURE FASEA1;
    BEGIN
    ......
    END;

   PROCEDURE FASEA2;
    BEGIN
    ......
    END;

 BEGIN
  FASEA1;
  FASEA2
 END;
(*---------------------------------------*)
 PROCEDURE FASEB;

   PROCEDURE FASEB1;
    BEGIN
```

```
        ......
        END;

       PROCEDURE FASEB2;
        BEGIN
        ......
        END;

     BEGIN
      FASEB1;
      FASEB2
     END;
   (*--------------------------------------*)
    PROCEDURE FASEC;
     BEGIN
     ......
     END;
   (*--------------------------------------*)
   BEGIN  (* Aquí empieza el programa *)
    FASEA;
    FASEB;
    FASEC
   END.
```

Supongamos ahora que las fases A1 y B1 fueran idénticas. Con esta estructura de programa no habría más remedio que tener los dos procedimientos repetidos, pues si dejáramos, por ejemplo, sólo FASEA1, no podría ser utilizado desde FASEB, pues aquél es un procedimiento local de FASEA y, por tanto, de su uso exclusivo.

La regla para saber cuándo puede ser utilizado un procedimiento es la siguiente:

Sólo puede ser utilizado un procedimiento desde dentro de su inmediato poseedor tomado en su conjunto y siempre desde puntos que se encuentren por detrás de su cabecera.

Por tanto, FASEA1, en el primer ejemplo, sólo puede ser utilizado desde dentro de FASEA en su conjunto, o sea, desde el propio FASEA y desde FASEA2.

Escribamos ahora la fase común como un procedimiento FASEAYB1 independiente:

```
PROGRAM RELACIONES;
(*--------------------------------------*)
 PROCEDURE FASEAYB1; (* Sirve para las fases A1 y B1 *)
  BEGIN
```

```
  ......
  END;
(*-------------------------------------*)
 PROCEDURE FASEA;

   PROCEDURE FASEA2;
    BEGIN
    ......
    END;

 BEGIN
  FASEAYB1;   (* Utiliza el procedimiento común *)
  FASEA2
 END;
(*--------------------------------------*)
 PROCEDURE FASEB;

   PROCEDURE FASEB2;
    BEGIN
    ......
    END;

 BEGIN
  FASEAYB1;   (* Utiliza el procedimiento común *)
  FASEB2
 END;
(*--------------------------------------*)
 PROCEDURE FASEC;
  BEGIN
  ......
  END;
(*-------------------------------------*)
BEGIN  (* Aquí empieza el programa *)
 FASEA;
 FASEB;
 FASEC
END.
```

El inmediato poseedor de **FASEAYB1** es el programa principal, por lo que puede ser utilizado desde cualquier punto situado por detrás de su cabecera, esté o no dentro de un procedimiento o función.

Si llamamos a un procedimiento o función «hijo» de otro cuando es local suyo, tendríamos que **FASEB2** es «hijo» de **FASEB**, que a su vez sería «hijo» del programa principal. También podríamos decir que **FASEB** es «padre» de **FASEB1**.

Según esto, los procedimientos **FASEA** y **FASEB** serían «hermanos» entre sí.

Con esta definición de parentescos podríamos definir la regla de esta otra manera:

Un procedimiento o función puede utilizar a otro si este último es «hijo» suyo, «hermano», ascendiente o «hermano» de un ascendiente estando, además, su cabecera escrita por delante en el programa.

Así, se podría utilizar al «hermano» del «padre» del «padre» pero no al «hijo» del «hijo».

Como FASEA1 es «hijo» de «hermano» de FASEB, no puede ser utilizado por éste.

Sin embargo, como FASEAYB1 es un procedimiento «hermano» y está por delante de FASEA y FASEB, puede ser utilizado por ambos.

La cabecera debe encontrarse por delante del punto de utilización porque una de las características del lenguaje PASCAL es que, sea cual sea el punto en el que aparece un identificador, éste debe haber sido definido previamente. Por ejemplo, el programa puede utilizar una variable que ha sido definida previamente en la zona VAR utilizando un tipo que ha sido definido previamente en la zona TYPE como subrango delimitado por dos constantes definidas previamente en la zona CONST. A la hora de escribir un programa esto no es una gran limitación, pues no tiene demasiado sentido utilizar algo que todavía no se ha definido.

Por tanto, la respuesta a la pregunta que se planteaba al principio (¿va a poder seguir siendo utilizada por el procedimiento?) es SI, siempre que MAYUSCULA esté escrita por delante de READLNDIA:

```
FUNCTION  MAYUSCULA(C: CHAR): CHAR;
 BEGIN
  IF ( 'a' <= C ) AND ( C <= 'z' )   (* Si C está entre a y z *)
    THEN
      MAYUSCULA:= CHR(ORD(C)-ORD('a')+ORD('A'))
    ELSE
      MAYUSCULA:=C
 END;

PROCEDURE READLNDIA (VAR D: DIADELASEMANA );
 VAR C: CHAR;
BEGIN
 READLN (C);
 C:= MAYUSCULA (C);  (* Hacemos a C mayúscula *)
 IF C='L' THEN  D:=LUNES     ELSE
 IF C='M' THEN  D:=MARTES    ELSE
 IF C='X' THEN  D:=MIERCOLES ELSE
 IF C='J' THEN  D:=JUEVES    ELSE
 IF C='V' THEN  D:=VIERNES   ELSE
 IF C='S' THEN  D:=SABADO    ELSE
 IF C='D' THEN  D:=DOMINGO
END;
```

Evidentemente, de esta manera READLNDIA pierde su «autosuficiencia» con la compensación de tener a MAYUSCULA disponible para su uso por otras partes del programa.

Para terminar, otra pregunta se plantea: si, además de esta función MAYUSCULA independiente, READLNDIA siguiera teniendo su propia función local, ¿cuál de ellas utilizaría el procedimiento?

Al igual que sucede con las variables, el PASCAL cuando encuentra un nombre de procedimiento o función busca primero entre los locales del punto en que se encuentra, por lo que READLNDIA utilizaría su propia función y el resto del programa utilizaría la independiente.

Así, las ventajas principales de ubicar unos procedimientos dentro de otros son dos:

Por un lado, es posible hacer a éstos últimos «autosuficientes», en el sentido de que contienen todas las funciones, procedimientos y variables necesarios para su funcionamiento, de manera que baste con copiarlos para utilizarlos en otros programas. (Aunque, en el caso concreto de READLNDIA, el tipo DIADELASEMANA debe estar definido en el programa principal.)

Por otro, permite sustituir los procedimientos generales, disponibles para todo el programa (entre los que se incluyen los predefinidos), por los suyos propios.

Con los procedimientos y funciones de uso general, sin embargo, lo más cómodo será ponerlos como «hijos» del programa principal para así poderlos utilizar desde cualquier punto.



PROCEDIMIENTOS RECURSIVOS

Supongamos que hubiera que programar la función «Factorial». El factorial de un número entero N se escribe como N! y se calcula multiplicando N por N-1 por N-2 ... hasta llegar a 1. Por ejemplo:

1! es igual a 1.
2! es igual a 2 por 1 igual a 2.
3! es igual a 3 por 2 por 1 igual a 6.
4! es igual a 4 por 3 por 2 por 1 igual a 24.
5! es igual a 5 por 4 por 3 por 2 por 1 igual a 120.

Como se puede ver, el factorial de un número N es igual a él mismo multiplicado por el factorial del número anterior.

2! es igual a 2 * 1!
3! es igual a 3 * 2!
4! es igual a 4 * 3!
5! es igual a 5 * 4!

Por ello, el programa para calcular el factorial de, digamos, 10, sería:

1. Calcular el factorial de 9.
2. Multiplicarlo por 10.

A su vez, el punto 1 (calcular el factorial de 9) sería:

A. Calcular el factorial de 8.
B. Multiplicarlo por 9.

Y así hasta llegar a «calcular el factorial de 1» que consistiría simplemente en tomar un 1. Un programa **PASCAL** con esta estructura debería tener una función para calcular el factorial de 10, otra para el de 9, etc.

Evidentemente, es una forma poco práctica de programar (imaginemos cómo sería para el factorial de 20, 30 ...). Sería más interesante tener una función única a la que se pasara como parámetro el número N. El programa para calcular el factorial de N sería:

«Calcular el factorial del parámetro transferido»:

— Si el parámetro vale 1, devolver 1, y si no:
   1. Calcular el factorial del parámetro transferido menos 1.
   2. Multiplicarlo por el propio parámetro y devolverlo.

Y para ejecutar el punto 1 se utilizaría exactamente el mismo programa, pero pasándole como parámetro N-1. Sólo en el caso de transferir 1 el programa se limitaría a tomar un 1 como resultado.

Esto es lo que se denomina un programa recursivo, pues en la descripción de sus pasos se hace referencia (o se utiliza) a sí mismo.

El **PASCAL** permite la escritura de procedimientos y funciones recursivos sin ninguna complicación adicional:

```
PROGRAM RECURSIVO;
 VAR NUMERO: INTEGER;

 FUNCTION FACTORIAL (N: INTEGER): REAL;
  BEGIN
   IF N <= 1 THEN FACTORIAL:=1.0
    ELSE
     FACTORIAL:= FACTORIAL (N-1) * N
  END;

BEGIN
 WRITE ('Número? '); READLN (NUMERO);
 WRITE ('Su factorial vale: ',FACTORIAL (NUMERO));
END.
```

Se ha utilizado el tipo REAL para poder obtener resultados mayores de los posibles con el tipo INTEGER. (El factorial de cero vale 1 y, por ello, se ha utilizado la comparación «menor o igual».)

Vamos a explicar de manera resumida cómo es posible que una función (o procedimiento) se llame a sí misma:

Imaginemos que se ejecuta FACTORIAL (3). En el momento de llamar a la función se toma una porción de memoria para la variable local N en la que se guarda un 3.

Posteriormente, y ya dentro de la función, se ejecuta FACTORIAL (2); entonces se toma una nueva porción de memoria para N y en ella se guarda un 2 (como todavía no se ha regresado al programa principal, la porción que se tomó para guardar el 3 sigue ocupada).

Al ejecutarse FACTORIAL (2), cada vez que aparece N, el PASCAL busca la más reciente porción de memoria con ese nombre, que es la que alberga un 2.

Una vez más, se ejecuta FACTORIAL (1) tomando otra nueva porción de memoria para N, en la que se guarda un 1. Debido a la instrucción IF, se devuelve 1.0 y se acaba la ejecución de FACTORIAL (1). Entonces se libera la memoria en que se guardó el 1 y se vuelve al punto desde el que se llamó, es decir, en medio de la ejecución de FACTORIAL (2).

Allí, el resultado de FACTORIAL (1) se multiplica por el valor de N que, al haberse liberado la memoria que guardaba el 1, es de nuevo 2. Tras esto se acaba la ejecución de FACTORIAL (2) recuperándose la memoria en que se guardó el 2 y devolviéndose 2*1.0 al punto medio de FACTORIAL (3).

Entonces 2.0 se multiplica por 3 (pues la más reciente porción de memoria con el nombre N es la que contiene un 3) y se devuelve 6.0 como resultado de FACTORIAL (3), quedando libre la memoria en que se guardó el 3.

Por supuesto, la posibilidad de escribir procedimientos recursivos se debe, en gran parte, a la existencia de variables locales (es decir, aquellas que se crean en el momento de llamarse a un procedimiento y cuya memoria queda libre tras ejecutarse éste).

Las necesidades de memoria pueden llegar a ser grandes, pues la cantidad utilizada para variables locales aumenta cada vez que se profundiza en la recursión. Así, con FACTORIAL (30) se llegaría a tener en un momento dado 30 variables N.

Por ello, al escribir un procedimiento semejante debe comprobarse que no se va a seguir llamando a sí mismo indefinidamente. En el caso de FACTORIAL eso se consigue con la instrucción IF.

Hay muchos problemas de proceso de datos cuya respuesta se plantea de forma recursiva y ahí el empleo de procedimientos recursivos simplifica enormemente la programación.

Sin embargo, debido a la mayor cantidad de memoria que emplean, de-

ben utilizarse con precaución y sólo en los casos en que una programación no recursiva sea poco adecuada o impracticable.

La función factorial se puede calcular de manera no recursiva mucho más eficientemente utilizando un bucle para multiplicar N por N-1 por N-2 ... por 1:

«Calcular el factorial del parámetro transferido»:

1. Guardar 1.0 en la variable F.
2. Para I variando su valor desde 2 hasta el parámetro hacer:
   — Guardar en F su anterior valor multiplicado por el valor de I.
3. Devolver el valor de F.

En el programa bastaría con sustituir la parte de la función por:

```
FUNCTION FACTORIAL (N: INTEGER): REAL;
  VAR I: INTEGER; F: REAL;
 BEGIN
  F:=1.0;
  FOR I:=2 TO N DO F:=F*I;
  FACTORIAL:=F
 END;
```



DECLARACION ANTICIPADA DE PROCEDIMIENTOS

Existe la posibilidad de una recursión encubierta «a varias bandas»:

```
PROCEDURE AAA;

 PROCEDURE BBB;
  BEGIN
   AAA
  END;

BEGIN
 BBB
END;
```

Al ejecutarse AAA, éste utiliza a BBB, que a su vez utiliza a AAA, etc.

(BBB está, lógicamente, dentro del inmediato poseedor de AAA y, por ello, lo puede utilizar).

Ligeramente distinto sería este otro caso de "dos bandas".

```
PROCEDURE BBB;
 BEGIN
  AAA
 END;

PROCEDURE AAA;
 BEGIN
  BBB
 END;
```

En principio, no sería posible compilar este programa, pues sólo se puede utilizar AAA desde detrás de su cabecera, y si se permutasen las posiciones, el problema se plantearía con BBB.

El PASCAL permite superar este problema mediante la descripción anticipada de la cabecera de un procedimiento.

Para ello, en un lugar suficientemente adelantado del programa se escribe la cabecera completa seguida de la palabra reservada FORWARD separada por un punto y coma.

Posteriormente, y ya donde esté escrito el procedimiento, se repite la cabecera sin poner la descripción de la lista de parámetros (si los hay) ni el tipo, si es una función.

```
PROCEDURE AAA; FORWARD;

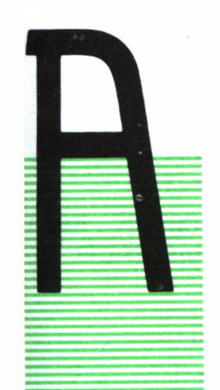
PROCEDURE BBB;
 BEGIN
  AAA
 END;
PROCEDURE AAA;
 BEGIN
  BBB
 END;
```

De esta manera, cuando el PASCAL encuentra la llamada a AAA dentro del procedimiento BBB, ya sabe que es un procedimiento y que, en este caso, no tiene parámetros.

EJEMPLOS CON PRODECIMIENTOS 9

ORDENACION DE DATOS





A menudo se plantea el problema de tener que ordenar, según un criterio dado, un conjunto de datos del mismo tipo. Por ejemplo, puede hacer falta colocar por orden alfabético los apellidos de los alumnos de un grupo, u ordenar de mayor a menor las notas de un examen.

Vamos a resolver este último problema por el método conocido como de selección directa. Consiste en lo siguiente:

Supongamos que tenemos 10 notas guardadas en TABLA, una variable de tipo ARRAY.

En primer lugar, hay que buscar la mayor nota de todas; una vez encontrada se intercambia con la que ocupaba la primera posición de TABLA. Tras este proceso, TABLA contiene las mismas notas que al principio, sólo que con la mayor de todas en primera posición.

A continuación se repite el mismo proceso, pero con las 9 notas restantes. Es decir, se busca la mayor de esas 9 (que están en las posiciones 2 a 10) y una vez encontrada se intercambia con la que estaba en la posición 2. Tras esto, TABLA sigue conteniendo la nota mayor en la posición 1 y, además, la segunda mayor en la posición 2.

Este proceso se repite para las posiciones 3 a 10 (con lo que queda la tercera nota en la posición 3), 4 a 10, etc., hasta llegar a hacerlo con las posiciones 9 a 10, momento en que queda TABLA definitivamente ordenada.

Por ejemplo, si las notas fueran:



al buscar la mayor de las diez se encuentra 9 en la cuarta posición; entonces se intercambia con la de la primera posición, 2:

9	/	7	6	2	3	1	8	4	7	5

Ahora se repite el proceso con todas menos la primera, permutándose, por tanto, el primer 7 con el 8 y llegándose a:

9	8	/	6	2	3	1	7	4	7	5

A continuación se repite con todas menos las dos primeras, etc., hasta tener TABLA ordenada. Las diferentes situaciones por las que pasaría TABLA serían:

9	8	7 /	2	3	1	6	4	7	5
9	8	7	7 /	3	1	6	4	2	5
9	8	7	7	6 /	1	3	4	2	5
9	8	7	7	6	5 /	3	4	2	1
9	8	7	7	6	5	4 /	3	2	1 (*)
9	8	7	7	6	5	4	3 /	2	1 (*)
9	8	7	7	6	5	4	3	2 /	1

Si utilizamos la variable I para indicar qué nota se está buscando (primera, segunda ...) y TOTAL para indicar el total de notas, la estructura queda así:

— Para I variando su valor desde 1 hasta TOTAL-1 hacer:
  — Buscar la mayor nota de las comprendidas entre las posiciones I y TOTAL y permutarla con la de la posición I.

Cuando la mayor entre I y TOTAL es precisamente la de la posición I, no es necesario permutarla consigo misma; son las situaciones marcadas con (*) en el ejemplo. Por tanto, la estructura quedaría mejor de esta otra manera:

— Para I variando su valor desde 1 hasta TOTAL-1 hacer:
  — Buscar la mayor nota de las comprendidas entre las posiciones I y si no es la de la posición I, permutarla con ella.

Si añadimos las instrucciones necesarias para leer datos de teclado y presentarlos en pantalla, tenemos el siguiente programa:

```
PROGRAM ORDENAR;

CONST
 MAX = 100;
TYPE
 TIPONOTAS = ARRAY [1..MAX] OF REAL;  (* MAX notas como mucho *)
VAR
 NOTAS     : TIPONOTAS;
 TOTALNOTAS: INTEGER;
(*----------------------------------------------------------*)
PROCEDURE LEEDATOS (VAR TABLA: TIPONOTAS; VAR TOTAL: INTEGER);
(* Pide datos y los guarda en la primera variable de la lista   *)
(* El número de datos introducidos lo devuelve en la segunda    *)

  VAR I: INTEGER; OK: BOOLEAN;
 BEGIN
  WRITE   ('Número de notas (de 1 a ',MAX,'): ');
  REPEAT
   READLN (TOTAL);
   OK:= (1<=TOTAL) AND (TOTAL<=MAX);
   IF NOT OK THEN WRITE ('No vale. Repita:')
  UNTIL OK;

  WRITELN ('Comience a introducirlas.');
  FOR I:=1 TO TOTAL DO
    BEGIN  WRITE ('Nota ',I:3,' : '); READLN (TABLA[I])  END;
 END;

(*----------------------------------------------------------*)
PROCEDURE PRESENTA (TABLA: TIPONOTAS; TOTAL: INTEGER);
(* Presenta el contenido de TABLA  *)

  VAR I: INTEGER;
 BEGIN
  FOR I:=1 TO TOTAL DO  WRITELN ('Nota ',I:3,' = ',TABLA[I]:4:1)
 END;

(*----------------------------------------------------------*)
PROCEDURE ORDENA (VAR TABLA: TIPONOTAS; TOTAL: INTEGER);
(* Ordena el contenido de la primera variable de la lista *)

  VAR I,J,INDICEMAYOR: INTEGER;  MAYOR: REAL;
 BEGIN
  FOR I:=1 TO TOTAL-1 DO
   BEGIN
```

```
(*-------------------------------------------------*)
(* Buscar la mayor de entre I y TOTAL.              *)
(* En principio se toma como mayor la de índice I *)
(* y luego se exploran las siguientes:             *)
(*-------------------------------------------------*)
 MAYOR:= TABLA[I];
 INDICEMAYOR:=I;
 FOR J:=I+1 TO TOTAL DO
     IF TABLA[J] > MAYOR THEN   (*  <------  COMPARACION *)
        BEGIN
          (* La mayor por ahora pasa a ser la de índice J *)
          MAYOR:= TABLA[J];
          INDICEMAYOR:=J
          END;


  (*-------------------------------------------------*)
  (* Si la mayor no es la de índice I, se permuta   *)
  (* con ésta.                                        *)
  (*-------------------------------------------------*)
   IF INDICEMAYOR <> I THEN
     BEGIN
      TABLA [INDICEMAYOR]:=TABLA[I];
      TABLA[I]:=MAYOR
     END
  END
END;


(*-------------------------------------------------------------*)
BEGIN
 LEEDATOS (NOTAS,TOTALNOTAS);
 ORDENA   (NOTAS,TOTALNOTAS);
 WRITELN ('------------------------------');
 PRESENTA (NOTAS,TOTALNOTAS)
END.
```

Si en lugar del test «TABLA[J] > MAYOR» se hubiera utilizado «TABLA[J] < MAYOR», las notas acabarían ordenadas de menor a mayor, pero entonces sería conveniente cambiar los nombres de MAYOR e INDICEMAYOR por MENOR e INDICEMENOR, respectivamente, para que resultaran coherentes.

Si se tratara de poner por orden alfabético una serie de palabras, el procedimiento sería similar. Supongamos que las palabras se guardasen en va-

riables de tipo **ARRAY OF CHAR**; entonces la tabla de palabras sería un **ARRAY OF** variables de ese tipo:

```
PROGRAM ORDENAPALABRAS;

CONST
 MAX = 100;  (* Máximo número de palabras *)
 LONG= 10;   (* Caracteres por palabra *)
TYPE
 TIPOPALABRA = ARRAY [1..LONG] OF CHAR;
 TIPOTABLA   = ARRAY [1..MAX] OF TIPOPALABRA;
VAR
 PALABRAS     : TIPOTABLA;
 TOTALPALABRAS: INTEGER;
 ......................
```

El procedimiento **ORDENA** sería similar, pero jugando con palabras en lugar de con números reales:

```
PROCEDURE ORDENA (VAR TABLA: TIPOTABLA; TOTAL: INTEGER);
(* Ordena el contenido de la primera variable de la lista *)

  VAR I,J,INDICEPRIMERA: INTEGER;  PRIMERA: TIPOPALABRA;
 BEGIN
  FOR I:=1 TO TOTAL-1 DO
   BEGIN
   (*--------------------------------------------*)
   (* Buscar la primera palabra de entre I y TOTAL.   *)
   (* En principio se toma por primera la de índice I *)
   (* y luego se exploran las siguientes:             *)
   (*--------------------------------------------*)
    PRIMERA:= TABLA[I];
    INDICEPRIMERA:= I;
    FOR J:=I+1 TO TOTAL DO
        IF ANTES (TABLA[J], (* que *) PRIMERA) THEN
           BEGIN
           (* La primera por ahora pasa a ser la de índice J *)
            PRIMERA:= TABLA[J];
            INDICEPRIMERA:=J
           END;
   (*--------------------------------------------*)
   (* Si la primera no es la de índice I, se permuta *)
   (* con ésta.                                      *)
   (*--------------------------------------------*)
    IF INDICEPRIMERA <> I THEN
```

```
      BEGIN
        TABLA [INDICEPRIMERA]:=TABLA[I];
        TABLA[I]:=PRIMERA
      END
    END
  END;
```

Como se ve, el procedimiento es prácticamente idéntico al de las notas, solo que adaptado al nuevo tipo de variables y con algunos nombres distintos. Mientras que con las notas bastaba con la comparación «TABLA[J] > MAYOR» para saber qué nota iba por delante, para comparar dos palabras se utiliza la función ANTES.

Esta función compara dos palabras letra a letra hasta encontrar el primer par de letras distintas entre sí. Una vez localizadas mira a ver cuál va antes alfabéticamente. Se podría escribir así (y siempre por delante de ORDENA):

```
FUNCTION ANTES ( A,B: TIPOPALABRA): BOOLEAN;
(* Devuelve TRUE si A va por delante de B alfabéticamente *)

 VAR I: INTEGER;
BEGIN
 (*-------------------------------------------*)
 (*  Se exploran las dos palabras de izquierda a  *)
 (*  derecha buscando la primera letra distinta:  *)
 (*-------------------------------------------*)
 I:=0;
 REPEAT I:=I+1 UNTIL ( A[I] <> B[I] ) OR ( I=LONG );
 (*---------------------------------------------*)
 (*  Una vez encontradas ( o llegado hasta el final) *)
 (*  se comparan para saber qué palabra va antes:    *)
 (*---------------------------------------------*)
 ANTES:= ( A[I] < B[I] )
 (*---------------------------------------------*)
 (*  Si A y B fueran iguales, se estaría comparando  *)
 (*  las dos últimas letras de cada una y como son   *)
 (*  iguales, ANTES valdría FALSE.                   *)
 (*---------------------------------------------*)
END;
```

Si las palabras se hubiesen guardado en variables de tipo STRING y no ARRAY OF CHAR, se habría podido poner directamente TABLA[J] < PRI-

MERA en lugar de ANTES (TABLA[J],PRIMERA) (con la mayoría de los compiladores de PASCAL).

En definitiva, el procedimiento ORDENA se puede adaptar a prácticamente cualquier tipo de variable, con tal de disponer de alguna función que nos permita decidir de entre dos valores cuál va por delante.

Hay una gran variedad de métodos de ordenación, muchos de los cuales son más eficientes que éste, pero para cantidades moderadas de datos el método de selección es suficientemente rápido además de ser muy sencillo de programar.



LAS TORRES DE HANOI

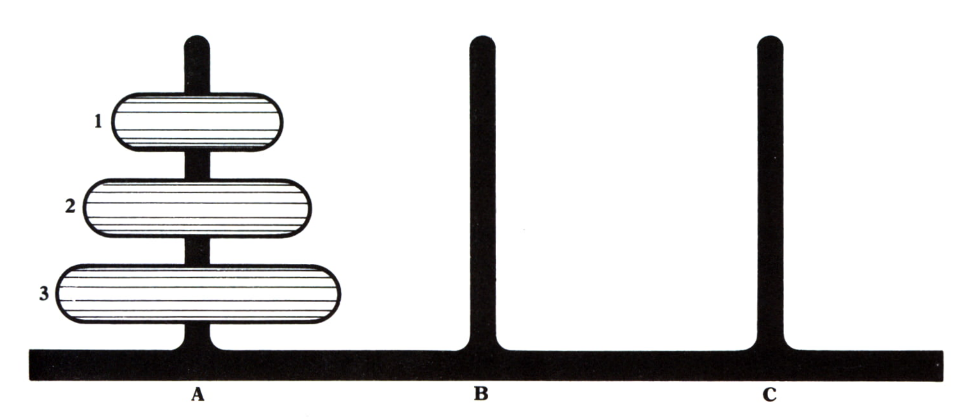
El juego de las *Torres de Hanoi* consiste en lo siguiente:

Se tiene una cantidad dada de piezas de plástico, papel o cualquier otro material de manera que sean todas de forma similar, pero distinto tamaño y que se puedan apilar. Por otra parte, existen tres lugares donde amontonarlas, llamémosles A, B y C.

Inicialmente las piezas se encuentran todas apiladas por orden de tamaño en el sitio A, de manera que la mayor es la que se encuentra debajo de todas y la menor la última de todas.

Se trata de pasar todas las piezas del sitio A al sitio C, moviéndolas de una en una y sabiendo que nunca se puede poner una pieza sobre otra que sea más pequeña. El sitio B se puede utilizar cuantas veces sea necesario para dejar piezas temporalmente. El jugador que consigue hacerlo en menor número de movimientos gana.

Habitualmente se juega con aros y con tres palos verticales donde ensartarlos para formar los montones. La situación de comienzo para una partida con tres aros sería, por tanto:



Para pasar el montón de tres a C hay que pasar primero los dos más pequeños a B, pasar entonces el disco mayor a C y luego pasar los otros dos de B a C.

A su vez, para pasar los dos más pequeños de A a B, por ejemplo, hay que pasar primero el menor a C, mover el mediano a B y, por fin, poner el más pequeño en B.

En general para mover un montón de N aros de un lugar de origen a otro de destino el procedimiento sería el siguiente:

«Mover N aros desde origen a destino»:

1. Mover el montón de N-1 aros que hay encima del número N desde origen al otro lugar que no es el destino.
2. Mover el aro número N desde origen a destino.
3. Mover el montón de N-1 aros desde el lugar en que se encuentra al destino.

A su vez los puntos 1 y 2 constan exactamente de los mismos pasos, pero con un valor distinto de N y distinto origen y destino. Sólo cuando el montón fuera de uno, se movería sin más directamente.

Como se ve este método de resolución es recursivo, pues se hace referencia a sí mismo. La escritura del procedimiento en PASCAL es inmediata. Para simplificar el procedimiento se van a guardar las torres en memoria como 0, 1 y 2 en lugar de como A, B y C.

```
PROCEDURE MOVERMONTON (N,ORIGEN,DESTINO: INTEGER);
 VAR OTROSITIO: INTEGER;
BEGIN
 OTROSITIO:=3-ORIGEN-DESTINO;
 IF N=1 THEN PASAARO (1,ORIGEN,DESTINO)
  ELSE
   BEGIN
    MOVERMONTON (N-1,ORIGEN,OTROSITIO);
    PASAARO (N,ORIGEN,DESTINO);
    MOVERMONTON (N-1,OTROSITIO,DESTINO)
   END
END;
```

PASAARO sería un procedimiento que, si el programa tuviera gráficos, se encargaría de desplazar en el dibujo el aro en cuestión. En nuestro caso nos vamos a limitar a indicar qué movimiento habría que hacer (suponiendo que el disco más pequeño es el 1 y que están numerados por orden).

Como las torres son 0, 1 y 2, su suma es siempre 3, por lo que dadas dos de ellas la tercera se obtiene restando las otras a 3.

El programa quedaría, por tanto, así:

```
PROGRAM TORRESHANOI;

 VAR TOTAL: INTEGER; (* Para guardar el total de aros *)

  PROCEDURE PASAARO (NUMERO,DESDE,HASTA: INTEGER);
  (* Indica el movimiento a hacer. Llama a las torres A, B y C *)
  BEGIN
  WRITE   ('Mover aro ',NUMERO,' desde ');
  WRITE   (CHR (DESDE + ORD('A')));
  WRITE   (' hasta ');
  WRITELN (CHR (HASTA + ORD('A')))
 END;

 PROCEDURE MOVERMONTON (N,ORIGEN,DESTINO: INTEGER);
  VAR OTROSITIO: INTEGER;
 BEGIN
  OTROSITIO:=3-ORIGEN-DESTINO;
  IF N=1 THEN PASAARO (1,ORIGEN,DESTINO)
   ELSE
    BEGIN
     MOVERMONTON (N-1,ORIGEN,OTROSITIO);
     PASAARO (N,ORIGEN,DESTINO);
     MOVERMONTON (N-1,OTROSITIO,DESTINO)
    END
 END;

BEGIN
 CLRSCR;
 WRITE ('Cuantos aros? '); READLN (TOTAL);

(* PINTARINICIO (TOTAL); Comentario hasta que alguien lo programe *)

 MOVERMONTON (TOTAL,0,2) (* Mover todas desde A hasta C *)
END.
```

Este programa indica los movimientos a hacer para conseguir mudar la torre en la menor cantidad posible de jugadas. A poco que se practique se verá que el método es sistemático, por lo que sería posible programarlo de manera no recursiva con alguna estructura de bucle. Sin embargo, dado que nunca se juega con muchos aros, la cantidad de memoria utilizada no es excesiva y la solución recursiva es mucho más clara.

Aunque los ejemplos de problemas recursivos presentados hasta ahora tienen una solución no recursiva bastante sencilla, no se debe pensar que siempre sucede así. Casos típicos en que la utilización de procedimientos

recursivos simplifica enormemente el trabajo son los programas de gestión de bases de datos y los propios programas compiladores.



BORRADO DE PANTALLA

Hasta ahora se ha supuesto que nuestra versión de PASCAL tiene el procedimiento predefinido CLRSCR o PAGE para el borrado de pantalla.

Si al copiar un programa éste utilizara el procedimiento CLRSCR y fuera PAGE el disponible, podríamos cambiar, en todos los puntos en que apareciera, un nombre por el otro. Sin embargo parece mejor solución no cambiar nada y añadir aquello que falte. Por ello, incorporaríamos al programa el siguiente procedimiento:

```
PROCEDURE CLRSCR;  BEGIN  PAGE  END;
```

Así, cada vez que apareciera CLRSCR se ejecutaría en realidad PAGE. Si fuera otro distinto bastaría con ponerlo dentro del procedimiento.

Si no hubiera nada predefinido al respecto habría que incorporar al procedimiento las instrucciones necesarias para el borrado. La manera más sencilla sería escribiendo suficientes líneas en blanco para hacer desaparecer todo lo que pudiera haber en la pantalla:

```
PROCEDURE CLRSCR;
(* Borra la pantalla *)
 CONST NUMLINEAS  =24;  (* Las líneas de la pantalla *)
       NUMCOLUMNAS=80;  (* Las columnas de una línea *)
 VAR FILA,COLUMNA: INTEGER;
BEGIN
 FOR FILA:=1 TO NUMLINEAS DO
  BEGIN
   FOR COLUMNA:=1 TO NUMCOLUMNAS DO WRITE(' ');
   WRITELN (* Para pasar a la siguiente línea *)
  END
END;
```

El lector probablemente encontrará mejores métodos a poco que profundice en las peculiaridades de su ordenador y de su lenguaje PASCAL.

En cualquier caso, el ejemplo sirve para ilustrar cómo es posible crear los procedimientos y funciones que falten para hacer funcionar programas escritos con otras versiones de PASCAL.

Otro ejemplo podría ser el procedimiento GOTOXY, que sirve para poner el cursor en un determinado punto de la pantalla para así poder escribir en la zona que nos interese. La cabecera sería:

```
PROCEDURE GOTOXY (X,Y: INTEGER);
(* Pone el cursor en la columna número X de la fila Y *)
```

Se deja al lector como ejercicio el escribir el resto del procedimiento.

MAS ESTRUCTURAS DE CONTROL Y TIPOS DE DATOS 10



CON las estructuras de control que ya conocemos somos capaces de hacer que un conjunto de instrucciones se pueda ejecutar en secuencia, de manera repetitiva o de manera selectiva, según un criterio dado. Con esto es posible, en teoría, construir cualquier programa. Sin embargo, los programas pueden ser simplificados por medio de la utilización de dos nuevas estructuras, CASE y GOTO, que veremos a continuación.

Además, vamos a ver un nuevo tipo de datos que permite trabajar con conjuntos de elementos y, por ejemplo, decidir si un elemento dado está o no en un conjunto: el tipo SET.



LA ESTRUCTURA CASE

Esta estructura podría considerarse como una estructura IF especial. IF, tal como la conocemos, permite escoger entre un máximo de dos posibilidades.

En multitud de ocasiones, sin embargo, es preciso escoger entre más de dos alternativas. Recordemos el procedimiento WRITEDIA:

```
PROCEDURE WRITEDIA (D: DIADELASEMANA );
 BEGIN
  IF D=LUNES     THEN WRITE('Lunes')     ELSE
  IF D=MARTES    THEN WRITE('Martes')    ELSE
  IF D=MIERCOLES THEN WRITE('Miércoles') ELSE
  IF D=JUEVES    THEN WRITE('Jueves')    ELSE
  IF D=VIERNES   THEN WRITE('Viernes')   ELSE
```

```
        IF D=SABADO    THEN WRITE('Sábado')    ELSE
                            WRITE('Domingo')
       END;
```

También se podría haber escrito agrupando los diferentes IF de esta otra manera:

```
IF D=LUNES     THEN WRITE('Lunes');
IF D=MARTES    THEN WRITE('Martes');
IF D=MIERCOLES THEN WRITE('Miércoles');
..........................
```

que es algo menos eficiente, pero que funciona exactamente igual (si, por ejemplo, D fuera LUNES, tras escribir LUNES se volvería a comprobar si D es MARTES, etc., cosa que no sucede con el procedimiento en su forma original).

La estructura CASE permite escoger entre varias instrucciones, según sea el resultado de una expresión dada. El tipo resultante de esta expresión debe ser INTEGER, CHAR o cualquier otro tipo escalar creado por nosotros. WRITEDIA se escribiría así:

```
PROCEDURE WRITEDIA (D: DIADELASEMANA );
 BEGIN
  CASE D OF
    LUNES     : WRITE('Lunes');
    MARTES    : WRITE('Martes');
    MIERCOLES : WRITE('Miércoles');
    JUEVES    : WRITE('Jueves');
    VIERNES   : WRITE('Viernes');
    SABADO    : WRITE('Sábado');
    DOMINGO   : WRITE('Domingo')
   END
 END;
```

La expresión (en este caso una variable) cuyo valor controla la decisión a tomar se escribe entre las palabras reservadas CASE y OF.

Tras esto se escriben las diferentes instrucciones a escoger separadas entre sí por punto y coma. Además, delante de cada una y separada por dos puntos debe escribirse la lista de uno o más valores (separados por co-

mas) para los que debe ejecutarse esa instrucción. Un valor dado sólo puede aparecer en, a lo sumo, una lista.

Para terminar la estructura CASE se escribe la palabra reservada END, tras la que podrían venir otras instrucciones separadas por punto y coma.

Hay que considerar qué sucede cuando el resultado de la expresión no se encuentra en ninguna de las listas. En las primeras versiones de PASCAL se producía un mensaje de error y el programa se detenía. Las versiones posteriores simplemente no ejecutan ninguna de las instrucciones de la estructura CASE y pasan a lo que venga a continuación.

La mayoría de los compiladores de PASCAL actuales permiten, sin embargo, definir opcionalmente una instrucción alternativa que se ejecutaría en estos casos. Para ello se escribe esta instrucción antes de la palabra END final precedida de la palabra reservada ELSE u OTHERWISE, según el compilador que se utilice (estamos ante algo no estándar del PASCAL y de ahí la posibilidad de que no en todas las versiones de PASCAL se utilice la misma palabra). Veamos un ejemplo:

```
PROGRAM ANALIZATECLA;
 (* Analiza las diferentes teclas existentes.  *)
 (* No se cuenta con la Ñ.                     *)
 VAR C: CHAR;

FUNCTION  MAYUSCULA(C: CHAR): CHAR;
 BEGIN
  IF ( 'a' <= C ) AND ( C <= 'z' )   (* Si C está entre a y z *)
    THEN  MAYUSCULA:= CHR(ORD(C)-ORD('a')+ORD('A'))
    ELSE  MAYUSCULA:=C
 END;

BEGIN
 WRITELN ('Pulse F para terminar.');
 REPEAT
  WRITE('Tecla (y Return): '); READLN (C);  C:= MAYUSCULA (C);
  CASE C OF
    'A','E','I','O','U' :         WRITELN ('Es una vocal.');
    'B','C','D','F','G','H','J',
    'K','L','M','N','P','Q','R',
    'S','T','V','W','X','Y','Z' : WRITELN ('Es una consonante.');
    '0','1','2','3','4',
    '5','6','7','8','9'         : WRITELN ('Es una cifra.')
     ELSE  WRITELN ('No es ni letra ni cifra.')
    END
  UNTIL C='F';
  WRITELN ('Adiós.')
 END.
```

Igual que sucede con la estructura IF, las instrucciones a escoger pueden ser simples, como en el ejemplo, o de cualquier otro tipo (REPEAT, FOR...), incluso secuencias de instrucciones. Para terminar, veamos otro ejemplo:

```
PROGRAM CALCULADORA;
 VAR N1,N2: REAL;  OPCION: INTEGER;

PROCEDURE PIDENUMEROS (VAR A,B: REAL);
 (* Lee dos números de teclado *)
BEGIN
 WRITELN ('Primer  número: '); READLN (A);
 WRITELN ('Segundo número: '); READLN (B)
END;

BEGIN
 REPEAT
 (* Poner aquí CLRSCR; o PAGE; etc. para borrar la pantalla. *)
  WRITELN;
  WRITELN ('1 - Sumar dos números.');
  WRITELN ('2 - Restarlos.');
  WRITELN ('3 - Multiplicarlos.');
  WRITELN ('4 - Dividirlos.');
  WRITELN ('5 - Obtener la raíz cuadrada.');
  WRITELN ('6 - Acabar programa.');
  WRITELN;
  WRITE ('Escoja opción: '); READLN (OPCION); WRITELN;

  CASE OPCION OF
    1: BEGIN
        PIDENUMEROS (N1,N2);
        WRITELN ('Su suma vale ',N1+N2)
       END;
    2: BEGIN
        PIDENUMEROS (N1,N2);
        WRITELN ('Su resta vale ',N1-N2)
       END;
    3: BEGIN
        PIDENUMEROS (N1,N2);
        WRITELN ('Su multiplicación vale ',N1*N2)
       END;
    4: BEGIN
        PIDENUMEROS (N1,N2);
        WRITELN ('Su división vale ',N1/N2)
       END;
    5: BEGIN
        WRITE('Número: ');  READLN (N1);
```

```
            IF N1<0.0 THEN WRITELN ('No valen números negativos.')
                      ELSE WRITELN ('Su raíz cuadrada vale ',SQRT(N1))
           END;
        6: BEGIN (* No hay que hacer nada *) END
       ELSE WRITELN ('OPCION NO VALIDA')
      END  (* Fin de CASE *)
     UNTIL OPCION=6
  END.
```

LA INSTRUCCION GOTO

La instrucción GOTO («ir a») hace que se continúe ejecutando el programa en otro punto distinto a aquél en que se encuentra en ese momento. Por decirlo de otra manera, permite «saltar» a cualquier punto del programa desde el lugar en que se encuentra la instrucción. Es equivalente a la instrucción GOTO de otros lenguajes como BASIC o FORTRAN.

El PASCAL tiene todas las estructuras de control necesarias para construir un programa, por lo que en multitud de ocasiones se le ha descrito de manera pobre y superficial como un «lenguaje para programar sin GOTO».

Sin embargo, hay casos (pocos) en que la utilización de GOTO puede simplificar un programa y por ello se contempla su uso en PASCAL. Estos casos suelen ser aquéllos en que, por algún suceso extraordinario, se desea cambiar la marcha normal de un programa. En cualquier caso, dada su gran potencia, debe utilizarse con gran cuidado y sólo en casos muy especiales.

La instrucción consta de la palabra reservada GOTO, seguida de la etiqueta del punto al que se desea transferir control. Esta etiqueta puede ser cualquier número entero de cuatro cifras como máximo, aunque algunas versiones de PASCAL permiten también el empleo de palabras o identificadores válidos. La etiqueta debe estar escrita justo antes de la instrucción a la que se desea saltar separada de ella por dos puntos.

Tan excepcional se considera su utilización que todas las etiquetas que se vayan a utilizar deben ser declaradas previamente tras la cabecera del programa (o procedimiento) y antes de la definición de datos de la siguiente manera:

```
PROGRAM EJEMPLOGOTO;
 LABEL              (* Label significa etiqueta *)
  10,20;
```

```
CONST
 PI=3.141592654;
VAR
 N: INTEGER;
BEGIN
 N:=0;
10: WRITELN ('Esto se escribe repetidas veces.');
 N:=N+1;
 IF  N < 4  THEN GOTO 10;
 GOTO 20;
 WRITELN ('Pero esto no.');
20: WRITELN ('Adiós.')
END.
```

es decir, tras la palabra reservada LABEL se escriben las diferentes etiquetas separadas entre sí por comas. Para terminar, se escribe un punto y coma.

Caso típico de utilización de GOTO es aquél en que se detecta un error en una fase temprana de la ejecución de un programa y se desea entonces que se detenga. Para conseguir esto último se podría hacer:

```
 ERROR1:= (...la condición de error que sea...)

 IF NOT ERROR1 THEN
   BEGIN
    (... AAA; BBB ... )

    ERROR2:= (...la condición de error que sea...)

    IF NOT ERROR2 THEN
      BEGIN
       (...el resto del programa...)
      END
   END;

 WRITELN ('Fin del programa.')
END.
```

Es decir, poner como condición para la ejecución de lo que viene a continuación la no existencia de error. Si, por ejemplo, el error se detectara

dentro de una estructura REPEAT, habría que poner también a éste como condición de salida del bucle:

```
        ERROR1:= (...la condición que sea...)
       UNTIL ERROR1 OR ....
```

Sin embargo, mediante GOTO el programa se simplifica:

```
     IF (...la condición de error que sea...) THEN GOTO 100;
     (... AAA; BBB ... )

     IF (...la condición de error que sea...) THEN GOTO 100;
     (...el resto del programa...)

  100: WRITELN ('Fin del programa.')
    END.
```

Una importante restricción a tener en cuenta es que sólo se puede saltar a un punto dentro del mismo bloque de programa en que nos encontremos en el momento del salto, es decir, no se puede ir desde dentro de un procedimiento a otro, o desde el programa principal a una función, etc. Por otra parte, el salto al interior de una instrucción estructurada (IF, FOR ...) desde fuera de ella puede producir errores inesperados:

```
GOTO 10; IF A THEN 10:WRITELN
```

Puede que alguien sienta la tentación de declarar al principio:

```
LABEL 10,20,30,40,50,60,70,80,90 ... 1000,1010,1020 ...
```

y programar «al estilo BASIC». En ese caso es que no ha comprendido el propósito y las ventajas de la programación estructurada. Sin embargo, tampoco se debe ser purista y complicarse la vida para no utilizar jamás la instrucción GOTO. La norma que hay que seguir en todo momento es la de la máxima claridad y, si ésta se consigue con GOTO, no hay que dudar en utilizarla.

Nota: La utilidad de acabar la ejecución de un procedimiento median-

te un salto al final es tal, que muchos compiladores disponen de una instrucción de salto especial para ello, que suele escribirse como EXIT (salida).



EL TIPO SET

Volvamos al programa CALCULADORA. Las opciones 1, 2, 3 y 4 utilizan todas, en primer lugar, la instrucción PIDENUMEROS (N1,N2). En lugar de esto se podría hacer el programa más corto si, antes de la instrucción CASE, se escribiera la única instrucción:

```
IF (1 <= OPCION) AND (OPCION <= 4) THEN PIDENUMEROS (N1,N2);
```

En otras palabras, llamar a **PIDENUMEROS** si **OPCION** es alguna del conjunto formado por 1, 2, 3 y 4.

Supongamos ahora que las opciones no fueran éstas, sino, por ejemplo, 1, 3, 4 y 5. Como ya no son consecutivas, para detectar que OPCION es alguna de ellas habría que utilizar un test mucho más complejo que además no valdría si en algún momento cambiásemos las opciones del programa que utilizan el procedimiento:

IF (OPCION=1) OR ((3 <= OPCION) AND (OPCION <= 5) THEN...

El PASCAL permite definir CONJUNTOS (sets en inglés) de elementos, de manera que se puedan utilizar con comodidad. Por ejemplo, los conjuntos formados por 1, 2, 3, 4 y 1, 3, 4, 5 se escribirían así:

[1,2,3,4] o [1,3,2,4] o [4,3,2,1] etc., para el primero, y
[1,3,4,5] o [1,5,4,3] etc., para el segundo

es decir, poniendo entre corchetes la lista de elementos que los forman separados por comas; da lo mismo el orden que se utilice. Cuando varios de los elementos están seguidos, como es el caso de todos los del primer conjunto y tres del segundo, es posible definirlos de manera abreviada:

[1..4] para el primero y [1,3..5] o [3..5,1] para el segundo

utilizando, por tanto, una notación similar a la de los subrangos.

Antes de entrar en detalles veamos una primera aplicación de los conjuntos: dado un elemento es posible saber si se encuentra en un conjunto utilizando el operador IN, que devuelve según el caso el resultado lógico TRUE (caso de encontrarse) o FALSE. Con este operador las dos instrucciones IF anteriores se escribirían de la siguiente manera:

```
IF OPCION IN [1..4]   THEN  PIDENUMEROS (N1,N2);
IF OPCION IN [1,3..5] THEN  ...
```

Es posible definir variables de tipo conjunto, es decir, variables donde poder guardar un conjunto de elementos. Por ejemplo:

```
VAR  ESPECIALES: SET OF 1..6;
     DIASLIBRES: SET OF DIADELASEMANA;
```

definiría las variables ESPECIALES y DIASLIBRES que sirven, respectivamente, para guardar cualquier conjunto de números entre 1 y 6 y cualquier conjunto de días de la semana. O sea, tras las palabras reservadas SET y OF se indica el tipo de elementos que pueden formar parte del conjunto, que debe ser siempre de tipo escalar o subrango de éstos. Como sucede con otros tipos, es posible definir antes el tipo conjunto:

```
TYPE  TIPOMENU= SET OF 1..6;
VAR   ESPECIALES: TIPOMENU;
```

Para guardar conjuntos en una variable se utiliza el operador de asignación de la manera conocida:

```
ESPECIALES:=[1..4];
DIASLIBRES:=[SABADO,DOMINGO];
```

Con estas asignaciones se podría escribir:

```
IF OPCION IN ESPECIALES  THEN  PIDENUMEROS (N1,N2);
IF NOT (HOY IN DIASLIBRES) THEN WRITELN ('Hoy hay que trabajar.');
```

Dependiendo del compilador, hay un límite para el máximo número de elementos que pueden tener los conjuntos; habitualmente es 256. Por ello, no se podría definir un SET OF INTEGER, pues un conjunto de estas características podría llegar a tener miles de elementos. El mayor conjunto del tipo TIPOMENU sería, sin embargo, [1,2,3,4,5,6] que tiene seis elementos.

En todo caso, el menor conjunto posible es el vacío, aquél que no tiene ningún elemento; se describe simplemente con dos corchetes: [].



Expresiones con conjuntos

Las operaciones entre conjuntos que dan como resultado otro conjunto son la unión, la intersección y la diferencia, que se indican, respectivamente, con los signos + , * y – . En caso de necesidad se pueden utilizar paréntesis.

La unión de dos conjuntos da como resultado otro del mismo tipo formado por los elementos de ambos:

```
['A','E'] + ['A','I']          da como resultado ['A','E','I'];
[1] + [2]                      da como resultado [1,2];
[] + [LUNES]                   da como resultado [LUNES];
```

La intersección, sin embargo, da el conjunto formado por los elementos comunes a ambos:

```
['A','E'] * ['A','I']          da como resultado ['A'];
[1,2] * [3,4]                  da como resultado [ ];
[LUNES] * [LUNES,MARTES]       da como resultado [LUNES];
```

Por último, la diferencia devuelve el conjunto formado por los elementos del primero que no se encuentran en el segundo:

```
['A','E'] - ['A','I']          da como resultado ['E'];
[LUNES,MARTES] - [LUNES]       da como resultado [MARTES];
```



Operaciones lógicas con conjuntos

Ya conocemos la prueba de pertenencia que se escribe poniendo en primer lugar el elemento (constante, variable o expresión) cuya pertenencia a un conjunto se desea comprobar, seguida de la palabra reservada IN tras la que viene el conjunto en cuestión (o variable o expresión de tipo conjunto). Formas correctas son:

LUNES IN (DIASLIBRES - [LUNES]) que daría como resultado FALSE y
1+2 IN ESPECIALES que daría como resultado TRUE.

Entre dos conjuntos se pueden utilizar las siguientes operaciones:

— Igualdad:

[1,2,3] = ([2,1] + [3]) es TRUE, mientras que
DIASLIBRES = [] es FALSE.

— Desigualdad:

[1,2,3] <> ([2,1] + [3]) es FALSE, mientras que
DIASLIBRES <> [] es TRUE.

— Inclusión (es decir, comprobar que todos los elementos de uno de ellos están en el otro):

['B','C'] <= ['A','B','C','D'] es TRUE (el primero está contenido en el segundo).

['A'..'Z'] >= ['0'] es FALSE (el segundo no está en el primero).
[ ] <= DIASLIBRES es TRUE.



Ejemplo

Supongamos que hay que formar equipos de fútbol con los alumnos de una clase. Los alumnos se encuentran numerados empezando por el 1. Vamos a escribir un programa al que se le tecleen los números de los alumnos de los diferentes equipos y que después nos diga qué partidos no son posibles.

Los equipos los guardaríamos en variables de tipo «conjunto de alumnos» pero, como puede haber varios, será una tabla de ellas lo que utilicemos:

```
PROGRAM EQUIPOS;
CONST
  MAX        = 100;    (* Como máximo 100 alumnos *)
  MAXEQUIPOS = 10;     (* Como máximo 10 equipos  *)
TYPE
  TIPOALUMNO = 1..MAX;
  TIPOEQUIPO = SET OF TIPOALUMNO; (* Conjuntos de alumnos *)
VAR
  CONJUNTO: ARRAY [1..MAXEQUIPOS] OF TIPOEQUIPO;
  TOTAL,              (* Para guardar el número de alumnos *)
  NUMEQUIP, I,        (* Para guardar el número de equipos *)
  ALUMNO,
  EQUIPO: INTEGER;
(*==============================================================*)
BEGIN
 CLRSCR; (* o PAGE etc. *)
 WRITE ('Número de alumnos: '); READLN (TOTAL);
 WRITE ('Número de equipos: '); READLN (NUMEQUIP);

FOR EQUIPO:=1 TO NUMEQUIP DO (* Formar los equipos de uno en uno *)
 BEGIN
  WRITELN ('Equipo número ',EQUIPO);
  CONJUNTO [EQUIPO]:=[];       (* En principio no tiene jugadores *)
  (*---------------------------------------------------*)
  (* Pedir los once jugadores e irlos añadiendo al equipo: *)
  (*---------------------------------------------------*)
  FOR I:=1 TO 11 DO
   BEGIN
  REPEAT (* Pedir el número de alumno hasta que sea correcto *)
    WRITE ('Alumno: ');
    READLN (ALUMNO);
    IF NOT (ALUMNO IN [1..TOTAL]) THEN WRITELN ('No vale.')
  UNTIL ALUMNO IN [1..TOTAL];
  CONJUNTO [EQUIPO] := CONJUNTO [EQUIPO] + [ALUMNO]
```

```
      END
     END;

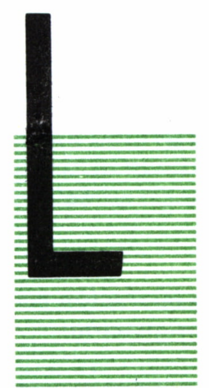
   (*-----------------------------------------*)
   (* Probar todos los emparejamientos posibles: *)
   (*-----------------------------------------*)
   FOR EQUIPO:=1 TO NUMEQUIP-1 DO
    FOR I:=EQUIPO+1 TO NUMEQUIP DO
       IF CONJUNTO [EQUIPO] * CONJUNTO [I] <> []  THEN
         WRITELN ('No puede jugar el equipo ',EQUIPO,
                    ' contra el equipo ',I)
   END.
```

Supongamos que hubiera 4 equipos. En este caso, los partidos posibles (en principio) serían:

— el 1 contra los equipos 2, 3 y 4
— el 2 contra los equipos 3 y 4
— el 3 contra el equipo 4

y de ahí los dos bucles FOR utilizados. Por otra parte, la condición para que un partido se pueda celebrar es que los dos equipos no tengan jugadores comunes.

REGISTROS 11



LOS registros o fichas (records, en inglés) se utilizan en multitud de actividades de la vida diaria. En un banco puede que se utilicen para guardar el nombre de cada cliente y el saldo de sus diferentes cuentas; en una central lechera, para el número, fecha y destino de los diferentes lotes de leche; en una biblioteca, para el título, autor, editorial y estante de cada libro, etc.

Actualmente cada vez se utilizan más los ordenadores para manejar este tipo de cosas. Sin ellos los bancos, por poner un ejemplo, serían incapaces de gestionar la enorme cantidad de cuentas que tienen hoy día.

Del PASCAL, por ahora, sólo conocemos el tipo ARRAY para guardar diferentes cosas en una misma variable, pero todas han de ser del mismo tipo; sin embargo, las fichas pueden tener información muy variada. El nombre del titular de una cuenta sería un dato del tipo ARRAY OF CHAR, mientras que el saldo sería de tipo REAL.

El PASCAL permite crear registros ajustados a nuestras necesidades. Las diferentes partes de un registro se denominan CAMPOS; así, las fichas de la central lechera tendrían tres campos para los tres diferentes datos de que constan. Al definir el tipo de ficha estos campos se deben describir de la siguiente manera:

TYPE FICHA = RECORD CAMPO1: TIPO1; CAMPO2: TIPO2 ... END

es decir, entre las palabras reservadas RECORD y END y separados por punto y coma se deben escribir los nombres de los diferentes campos seguidos de dos puntos y el tipo de dato que sean. Cuando dos campos o más son del mismo tipo, se puede ahorrar escritura poniendo sus nombres uno detrás de otro separados por comas, tras lo cual vendrían los dos puntos

y el tipo. (Por supuesto, es posible definir una variable poniendo a su lado directamente la descripción del tipo, pero siempre es mejor definir éste previamente.)

Los campos pueden ser de absolutamente cualquier tipo o subrango de tipo que esté previamente definido.

Supongamos que se desea guardar la fecha en una variable de tipo registro. Esta tendría tres campos: los de día y año, que serían números enteros, y el mes, que sería de tipo TIPOMES:

```
TYPE
 TIPOMES = (ENERO,FEBRERO,MARZO,ABRIL,MAYO,JUNIO,JULIO,AGOSTO,
            SEPTIEMBRE,OCTUBRE,NOVIEMBRE,DICIEMBRE);
 TIPOFECHA = RECORD
              DIAMES,ANYO : INTEGER;
              MES         : TIPOMES
             END;
VAR
 FECHADEHOY, NACIMIENTO: TIPOFECHA;
```

Como sucedía con las variables de tipo ARRAY, la única operación posible con todos los elementos de un registro a la vez es la asignación:

```
NACIMIENTO:=FECHADEHOY;
```

Esto copiaría los campos de FECHADEHOY, uno por uno, en los de NACIMIENTO.

Para hacer referencia a un campo en concreto de un registro se escribe, en primer lugar, el nombre del registro seguido de un punto y del nombre del campo:

```
WRITELN (FECHADEHOY.DIAMES + 1);
```

Como el campo DIAMES es de tipo INTEGER, FECHADEHOY.DIAMES se puede utilizar exactamente igual que cualquier variable INTEGER.

Por tanto, la parte de instrucciones del programa cuyos datos hemos descrito antes podría ser:

```
BEGIN
 FECHADEHOY.DIAMES:= 26;
 FECHADEHOY.MES   := AGOSTO;
 FECHADEHOY.ANYO  := 1986
END.
```



LA ESTRUCTURA WITH

Está claro que resulta muy incómodo tener que escribir el nombre del registro cada vez que se utiliza un campo, pero es inevitable para no confundir los campos de dos registros del mismo tipo. Si en una parte del programa sólo se hiciera referencia a un registro en concreto sería bueno poder decirle al compilador algo como: «Bueno, en toda esta zona sólo voy a trabajar con el registro Tal, por lo que discúlpame de poner su nombre cada vez.»

Esto se puede hacer de la siguiente manera:

Se escriben en primer lugar las palabras reservadas WITH y DO con el nombre del registro entre medias, y detrás la instrucción en la que se va a omitir su nombre. Si fueran varias, se enmarcan por delante y detrás con las palabras reservadas BEGIN y END, respectivamente:

```
BEGIN
 WITH FECHADEHOY DO  (* "Con FECHADEHOY haz:" *)
  BEGIN
   DIAMES:= 26;
   MES   := AGOSTO;
   ANYO  := 1986
  END
END.
```

En la mayoría de los compiladores la estructura WITH no sólo hace los programas más cortos y claros, sino que, además, permite hacerlos más rápidos.

El campo de un registro puede ser a su vez otro registro. Vamos a ver cómo podría empezar un programa que manejara las fichas de la central lechera:

```
PROGRAM LECHERO;
TYPE
 TIPOMES = (ENERO,FEBRERO,MARZO,ABRIL,MAYO,JUNIO,JULIO,AGOSTO,
           SEPTIEMBRE,OCTUBRE,NOVIEMBRE,DICIEMBRE);
 TIPOCIUDAD= ARRAY [1..12] OF CHAR; (* Para guardar la ciudad *)
 TIPOFECHA = RECORD
```

```
                DIAMES,ANYO : INTEGER;
                MES         : TIPOMES
               END;
  TIPOFICHA = RECORD
                LOTE    : INTEGER;
                FECHA   : TIPOFECHA;
                DESTINO: TIPOCIUDAD
               END;
VAR
 LOTEDEHOY: TIPOFICHA;

BEGIN
 WITH LOTEDEHOY DO  (* "Con LOTEDEHOY haz:" *)
  BEGIN
   LOTE          := 1234;
   (* FECHA es un registro con tres campos: *)
   FECHA.DIAMES:= 26;
   FECHA.MES    := AGOSTO;
   FECHA.ANYO   := 1986;
   DESTINO      := 'GUADALAJARA '    (* 12 letras *)
   END;
 ..............
```

Si no se utilizara WITH, habría que poner LOTEDEHOY.FECHA para referirse a la fecha y como ésta a su vez es un registro, habría que escribir LOTEDEHOY.FECHA.MES para el mes en concreto. Sin embargo, se podría poner otro WITH para FECHA:

```
WITH LOTEDEHOY DO  (* "Con LOTEDEHOY haz:" *)
 BEGIN
  LOTE     := 1234;
  WITH FECHA DO
   BEGIN
    DIAMES:= 26;
    MES   := AGOSTO;
    ANYO  := 1986
   END;
  DESTINO := 'GUADALAJARA '    (* 12 letras *)
 END;
```

o, de manera más clara:

```
WITH LOTEDEHOY DO WITH FECHA DO
 BEGIN
```

```
    LOTE    := 1234;
    DIAMES  := 26;
    MES     := AGOSTO;
    ANYO    := 1986;
    DESTINO := 'GUADALAJARA '    (* 12 letras *)
  END;
```

Tras WITH LOTEDEHOY DO sólo hay otra instrucción, WITH ...(que engloba a su vez a las demás), por lo que no se utiliza el par BEGIN/END.

Se pueden poner unos bloques WITH dentro de otros siempre que correspondan a registros de diferente tipo y que estén unos completamente dentro de otros.

Cuando el PASCAL encuentra una estructura WITH toma nota de la porción de memoria en que se encuentra el registro, y esa anotación es la que utiliza para los campos. Por ello, una instrucción como

```
I:=3;
WITH TABLA [I] DO
  FOR I:=1 TO 10 DO WRITELN (LOTE);
```

en que TABLA fuera un ARRAY OF TIPOFICHA siempre mostraría el número de lote de la ficha 3, que es la que aparecía junto a WITH al llegar ahí. Para presentar los diferentes lotes lo correcto sería:

```
FOR I:=1 TO 10 DO
 WITH TABLA [I] DO WRITELN (LOTE);
```

para que cada vez que se pase por WITH la ficha sea la adecuada.



REGISTROS VARIANTES

A veces el aspecto de un registro depende del valor de uno de los campos. Por ejemplo, en las fichas de personal de una empresa podrían estar los datos del cónyuge sólo si el estado civil fuera casado. En PASCAL es posible definir fichas con campos fijos y con otros que pueden variar según el valor de alguno de aquéllos.

Veamos un ejemplo:

```
TYPE
 TIPOFECHA = RECORD
              DIAMES,ANYO : INTEGER;
              MES : (ENERO,FEBRERO,MARZO,ABRIL,MAYO,JUNIO,JULIO,
                     AGOSTO,SEPTIEMBRE,OCTUBRE,NOVIEMBRE,DICIEMBRE)
             END;
 TIPONOMBRE= ARRAY [1..20] OF CHAR;
 TIPOFICHA = RECORD
              NOMBRE: TIPONOMBRE;
              EDAD  : 16..70;
              CASE ESTADOCIVIL: (SOLTERO,CASADO,DIVORCIADO,VIUDO) OF
               SOLTERO: ();
               CASADO : (CONYUGE: TIPONOMBRE; FECHABODA: TIPOFECHA);
               DIVORCIADO,VIUDO: (FINAL: TIPOFECHA)
             END;
```

Con esta estructura, todas las fichas tienen los campos NOMBRE, EDAD y ESTADOCIVIL. Además, y según el valor de éste último campo, tienen los campos CONYUGE y FECHABODA si es casado, y el campo FINAL cuando es viudo o divorciado. Nótese que algunos de los tipos utilizados han sido definidos sobre la marcha al definir el registro, mientras que otros lo han sido con anterioridad.

Los campos variables siempre deben estar al final del registro y a continuación del campo que decide la estructura (ESTADOCIVIL en este caso), utilizándose algo muy similar a la instrucción CASE, sólo que poniendo a continuación de cada posible valor, y entre paréntesis, la lista de campos específicos que le corresponden. Un nombre de campo no puede aparecer repetido en varias listas.

Para escribir el contenido de un registro semejante haríamos:

```
PROCEDURE PONFECHA (F: TIPOFECHA);
 (* Presenta la fecha con números: 23-11-86 etc. *)
 BEGIN
  WITH F DO WRITE (DIAMES:2,'-',ORD(MES)+1:2,'-',ANYO MOD 1900:2)
 END;

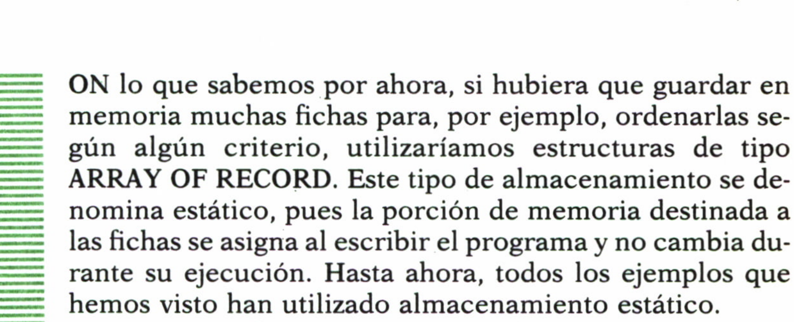
PROCEDURE PRESENTA (FICHA: TIPOFICHA);
 (* Presenta las diferentes partes de una ficha *)
BEGIN
WITH FICHA DO
 BEGIN
  WRITELN ('Nombre:        ',NOMBRE);
```

```
WRITELN ('Edad:           ',EDAD);
WRITE   ('Estado civil: ');
CASE ESTADOCIVIL OF
  SOLTERO: WRITELN ('Soltero');
  CASADO : BEGIN
             WRITELN ('Casado');
             WRITELN ('Nombre del cónyuge: ',CONYUGE);
             WRITE   ('Fecha matrimonio  : ');
             PONFECHA (FECHABODA);
             WRITELN
           END;
  DIVORCIADO: BEGIN
                WRITELN ('Divorciado');
                WRITE   ('Desde: ');
                PONFECHA (FINAL);
                WRITELN
              END;
  VIUDO  : BEGIN
             WRITELN ('Viudo');
             WRITE   ('Desde: ');
             PONFECHA (FINAL);
             WRITELN
           END
 END (* fin de CASE *)
END (* fin de WITH *)
END; (* fin del procedure *)
```

A pesar de tener diferentes campos según el valor de ESTADOCIVIL, la porción de memoria ocupada por una ficha cualquiera es siempre la misma e igual a la necesaria para el caso en que se precise más, que en este caso es cuando la ficha tiene los campos NOMBRE, EDAD, ESTADOCIVIL, CONYUGE y FECHABODA.

Algunas versiones de PASCAL existentes para ordenadores domésticos carecen de la posibilidad de crear registros variantes.

ALMACENAMIENTO EN MEMORIA DE REGISTROS 12



CON lo que sabemos por ahora, si hubiera que guardar en memoria muchas fichas para, por ejemplo, ordenarlas según algún criterio, utilizaríamos estructuras de tipo **ARRAY OF RECORD**. Este tipo de almacenamiento se denomina estático, pues la porción de memoria destinada a las fichas se asigna al escribir el programa y no cambia durante su ejecución. Hasta ahora, todos los ejemplos que hemos visto han utilizado almacenamiento estático.

Existe, no obstante, un método de almacenamiento en memoria denominado dinámico, que permite reservar porciones de memoria sobre la marcha e incluso utilizar una determinada zona para diferentes cometidos en diferentes momentos.

Vamos a escribir un programa para ordenar por orden alfabético de apellidos las fichas de los diferentes clientes de un banco, utilizando las cosas de **PASCAL** que conocemos por ahora. En primer lugar, las fichas se leerán y guardarán en una tabla; a continuación se procederá a ordenarlas y, por último, se mostrarán en pantalla.

Las fichas tendrán campos para un máximo de tres saldos, aunque, como se verá en seguida, el poner otros campos no supondría grandes cambios.

```
PROGRAM FICHAS;

CONST
 MAXNUM =100;  (* Máximo número de fichas admitido *)
 MAXLONG=20;   (* Máxima longitud de los nombres *)
TYPE
 TEXTO= ARRAY [1..MAXLONG] OF CHAR;
 FICHA= RECORD
        NOMBRE,APELLIDO: TEXTO;
```

```
          SALDO1,SALDO2,SALDO3: REAL
         END;
 TABLA= ARRAY [1..MAXNUM] OF FICHA;
VAR
 CLIENTES: TABLA;
 NUMERO,I  : INTEGER;
(*-----------------------------------------------------------*)
 FUNCTION ANTES ( A,B: TEXTO): BOOLEAN;
 (* Devuelve TRUE si A va por delante de B alfabéticamente *)
  VAR I: INTEGER;
 BEGIN
  I:=0;
  REPEAT I:=I+1 UNTIL ( A[I] <> B[I] ) OR ( I=MAXLONG );
  ANTES:= ( A[I] < B[I] )
 END;
(*-----------------------------------------------------------*)
PROCEDURE ORDENA (VAR T: TABLA; TOTAL: INTEGER);
(* Ordena el contenido de la tabla T *)

  VAR I,J,INDICEPRIMERA: INTEGER;  PRIMERA: FICHA;
 BEGIN
  FOR I:=1 TO TOTAL-1 DO
   BEGIN
   (*----------------------------------------------*)
   (* Buscar la primera ficha de entre I y TOTAL.     *)
   (* En principio se toma por primera la de índice I *)
   (* y luego se exploran las siguientes:             *)
   (*----------------------------------------------*)
    PRIMERA:= T [I];
    INDICEPRIMERA:= I;
    FOR J:=I+1 TO TOTAL DO      (* Comparar apellido: *)
        IF ANTES (T[J].APELLIDO, PRIMERA.APELLIDO) THEN
           BEGIN
           (* La primera por ahora pasa a ser la de índice J *)
            PRIMERA:= T [J];
            INDICEPRIMERA:=J
           END;
   (*---------------------------------------------*)
   (* Si la primera no es la de índice I, se permuta *)
   (* con ésta.                                      *)
   (*---------------------------------------------*)
    IF INDICEPRIMERA <> I THEN
      BEGIN
       T [INDICEPRIMERA]:=T [I];
       T [I]:=PRIMERA
      END
   END
 END;
(*-----------------------------------------------------------*)
```

```
PROCEDURE LEEDATOS (VAR F: FICHA);
BEGIN
 WITH F DO
  BEGIN
   WRITELN;
   WRITE ('Apellido: '); READLN (APELLIDO);
   WRITE ('Nombre   : '); READLN (NOMBRE);
   WRITE ('Cuenta 1: '); READLN (SALDO1);
   WRITE ('Cuenta 2: '); READLN (SALDO2);
   WRITE ('Cuenta 3: '); READLN (SALDO3)
  END
END;
(*-----------------------------------------------------------*)
PROCEDURE PRESENTADATOS (F: FICHA);
BEGIN
 WITH F DO
  BEGIN
   WRITELN;
   WRITELN ('Apellido: ',APELLIDO);
   WRITELN ('Nombre   : ',NOMBRE);
   WRITELN ('Cuenta 1: ',SALDO1:10:2);
   WRITELN ('Cuenta 2: ',SALDO2:10:2);
   WRITELN ('Cuenta 3: ',SALDO3:10:2)
  END
END;
(*-----------------------------------------------------------*)
BEGIN
 WRITELN ('Número de clientes (máximo ',MAXNUM,'): ');
 READLN (NUMERO);
 FOR I:=1 TO NUMERO DO  LEEDATOS (CLIENTES [I]);

 ORDENA (CLIENTES, NUMERO);
 WRITELN;
 WRITELN ('FICHAS YA ORDENADAS:');

 FOR I:=1 TO NUMERO DO  PRESENTADATOS (CLIENTES [I])
END.
```

Como se ve, si quisiéramos añadir más campos a las fichas, lo único que habría que cambiar, además de la descripción de las fichas, serían los procedimientos LEEDATOS y PRESENTADATOS. El procedimiento ORDENA es prácticamente igual al que ya se estudió en su momento.



ALMACENAMIENTO DINAMICO

El programa anterior, al utilizar almacenamiento estático, tiene el inconveniente de que el máximo número de fichas (100) está definido al crear el programa.

Para evitar que pueda quedarse corto hay que ponerlo suficientemente grande, pero entonces habrá ocasiones en que sólo se utilizará una pequeña parte de la memoria reservada al principio.

En PASCAL es posible crear variables no en el momento de escribir el programa, sino cuando éste se está ejecutando y a medida que se vayan necesitando. Al no estar definidas en la zona de declaración de datos, estas variables no tienen nombre y para ello se utiliza lo que se denomina PUNTEROS.

Un puntero es una variable especial que sirve para guardar una indicación de en qué sitio de la memoria se encuentra un registro. Si escribimos:

```
TYPE
  FICHA= RECORD
          NOMBRE,APELLIDO: TEXTO;
          SALDO1,SALDO2,SALDO3: REAL
         END;
  TIPOPUNT = ^FICHA;
VAR A,B: TIPOPUNT;
```

tendremos que todas las variables de tipo TIPOPUNT son punteros que sirven para «apuntar» a variables de tipo FICHA. Es decir, el tipo de puntero se indica con el símbolo ^ seguido del tipo de variable al que apunta.

Cuando el programa ya está funcionando, para reservar sitio para una ficha se utiliza la función NEW (nuevo, en inglés):

```
NEW (A);
```

de esta manera se reservaría memoria para una variable de tipo FICHA y su dirección quedaría guardada en la variable A. Cuando quisiéramos utilizar la ficha, en lugar del nombre que no tiene pondríamos A^, que significa «la variable apuntada por A» y es totalmente equivalente.

La única operación posible entre punteros es la asignación, es decir, guardar el contenido de uno en otro.

Para guardar muchas fichas necesitaríamos tener tantos punteros como fichas, por ejemplo, con un ARRAY [1..MAXNUM] OF TIPOPUNT, con lo que si esta tabla de punteros se llamara CLIENTES (la antigua ya no existe), para leer las fichas haríamos:

```
FOR I:=1 TO NUMERO DO
 BEGIN
```

```
    WRITELN;
    NEW (CLIENTES [I]); (* Reservar sitio para la ficha número I *)
    LEEDATOS (CLIENTES [I]^ )            (* Rellenarla con datos *)
   END;
```

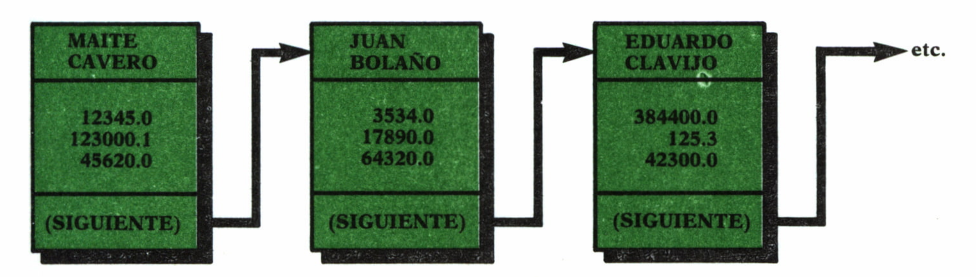
y de manera similar con el resto del programa principal y el procedimiento ORDENA. Este último, no obstante, podría ser mejorado ligeramente, pues para permutar dos fichas bastaría con permutar el contenido de sus dos punteros sin tocarlas a ellas para nada.

Esta manera de proceder sigue teniendo el mismo problema que la anterior, aunque menos grave, pues, al ocupar un puntero mucha menos memoria que una ficha, nos podríamos permitir preparar la tabla con un número sobradamente amplio de elementos.

Una solución mejor podría ser guardar el puntero de cada ficha en la anterior a ella, en un campo especialmente preparado para ello.

Para llegar a una ficha dada habría que tomar de la primera el puntero que alberga y con él podríamos utilizar ya la segunda ficha, de la que tomaríamos el puntero que lleva a la tercera, etc., hasta encontrar la ficha deseada.

De esta manera, tanto punteros como fichas se irían reservando en memoria según se fueran necesitando. Las fichas quedarían más o menos así:



Esto es lo que se denomina una estructura de tipo cola. Haría falta, además, un puntero aparte para utilizar el primer elemento. Para indicar que un elemento de la cola es el último lo que se hace es dar al puntero que alberga el valor predefinido NIL, cuyo significado es que no apunta a ninguna variable.

Veamos cómo se escribiría un programa que simplemente fuera guardando fichas para presentarlas luego:

```
  PROGRAM COLA;
  LABEL 10;
```

```
CONST MAXLONG=20;   (* Máxima longitud de los nombres *)
TYPE
 TEXTO= ARRAY [1..MAXLONG] OF CHAR;
 PUNTERO= ^FICHA;   (* Sirve para apuntar a fichas *)
 FICHA= RECORD
         NOMBRE,APELLIDO: TEXTO;
         SALDO1,SALDO2,SALDO3: REAL;
         SIGUIENTE: PUNTERO
        END;
VAR
 PRIMERO,            (* Apuntará al primer elemento de la cola *)
 ULTIMO,                (* y éste al último. *)
 P          : PUNTERO;
 FICHANUEVA: FICHA;
 FIN        : BOOLEAN;
(*-------------------------------------------------------------*)
PROCEDURE LEEDATOS (VAR F: FICHA);
BEGIN
 WITH F DO
  BEGIN
   WRITELN;
   WRITE ('Apellido: '); READLN (APELLIDO);
   WRITE ('Nombre   : '); READLN (NOMBRE);
   WRITE ('Cuenta 1: '); READLN (SALDO1);
   WRITE ('Cuenta 2: '); READLN (SALDO2);
   WRITE ('Cuenta 3: '); READLN (SALDO3)
  END
END;
(*-------------------------------------------------------------*)
PROCEDURE PRESENTADATOS (F: FICHA);
BEGIN
 WITH F DO
  BEGIN
   WRITELN;
   WRITELN ('Apellido: ',APELLIDO);
   WRITELN ('Nombre   : ',NOMBRE);
   WRITELN ('Cuenta 1: ',SALDO1:10:2);
   WRITELN ('Cuenta 2: ',SALDO2:10:2);
   WRITELN ('Cuenta 3: ',SALDO3:10:2)
  END
END;
(*-------------------------------------------------------------*)
BEGIN
 PRIMERO:= NIL;      (* Al principio no hay ni una ficha *)
 ULTIMO := NIL;

 (*-----------------------------------------------------------*)
 (* Leer primera ficha y reservarle sitio dejando su dirección *)
```

```
(* en el puntero PRIMERO y en ULTIMO (por ahora):                    *)
(*-------------------------------------------------------------------*)
WRITELN ('Introduzca 0 como apellido para terminar.');
LEEDATOS (FICHANUEVA);
IF FICHANUEVA.APELLIDO [1] <> '0' THEN
  BEGIN
   NEW (PRIMERO);
   PRIMERO^ := FICHANUEVA;          (* Guardar la ficha *)
   PRIMERO^.SIGUIENTE:= NIL;      (* No más fichas detrás *)
   ULTIMO   := PRIMERO;
  END
 ELSE GOTO 10;          (* Si no hay ni una ficha, acabar *)

(*-------------------------------------------------------------*)
(* Leer las siguientes fichas hasta que se dé apellido 0: *)
(*-------------------------------------------------------------*)
REPEAT
 LEEDATOS (FICHANUEVA);
 FIN:= (FICHANUEVA.APELLIDO [1] = '0');
 IF NOT FIN THEN
  BEGIN
  (*----------------------------------------------------------*)
  (* Reservar sitio dejando su dirección en la última *)
  (* ficha. ULTIMO pasa a apuntar a la nueva ficha.    *)
  (*----------------------------------------------------------*)
   NEW (ULTIMO^.SIGUIENTE);
   ULTIMO:= ULTIMO^.SIGUIENTE;
   ULTIMO^:= FICHANUEVA;            (* Guardar la ficha *)
   ULTIMO^.SIGUIENTE:= NIL        (* No más fichas detrás *)
  END
UNTIL FIN;

(*-----------------------------------------------------------*)
(* Presentar las fichas hasta llegar a la última.     *)
(* El puntero P va apuntando a las sucesivas fichas: *)
(*-----------------------------------------------------------*)
P:= PRIMERO;
WHILE P <> NIL DO (* Mientras que P apunte a una ficha...*)
 BEGIN
  PRESENTADATOS (P^);
  P:=P^.SIGUIENTE  (* Pasa a apuntar a la siguiente ficha *)
 END;

10: END.
```

Al definir el tipo PUNTERO se ha hecho referencia al tipo FICHA que todavía no estaba definido. Este es uno de los pocos casos en que esto está permitido. Si se hiciera primero la definición de FICHA, como en ésta se

hace a su vez referencia al tipo PUNTERO que todavía no estaría definido, se produciría un error.

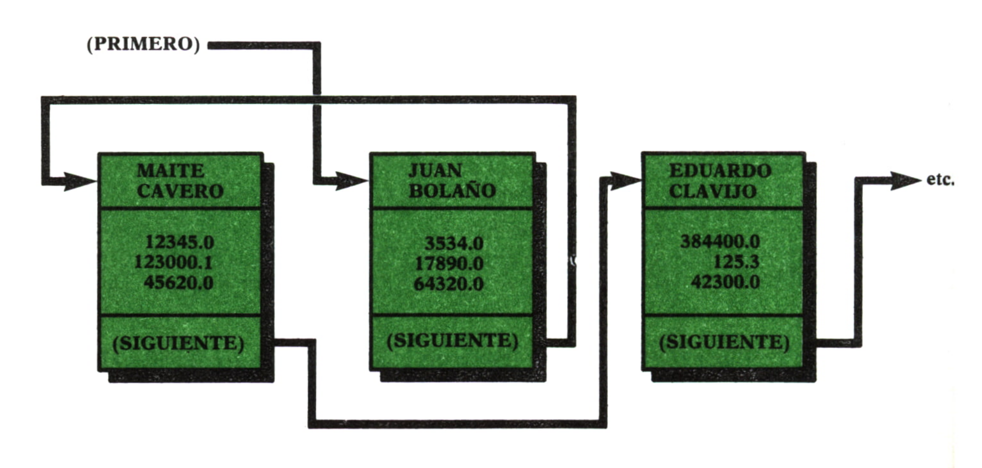
Sin embargo, algunos compiladores no toleran esta situación, por lo que habría que definir FICHA de la siguiente manera:

```
TYPE
 FICHA= RECORD
        NOMBRE,APELLIDO: TEXTO;
        SALDO1,SALDO2,SALDO3: REAL;
        SIGUIENTE: ^FICHA  (* Sirve para apuntar a otra ficha *)
       END;
```

y al definir los otros punteros se pondría también ^FICHA.

Desafortunadamente, hay compiladores con los que, como el campo SIGUIENTE y las variables no se han descrito utilizando la misma definición previa, son conceptuados como de distinto tipo y no es posible asignarlos unos a otros. En este caso, para guardar la dirección de la primera ficha, por ejemplo, se podría utilizar el campo SIGUIENTE de una ficha que llamaríamos ANTESQUEPRIMERA, cuyos otros campos se desperdiciarían.

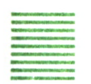
Para la ordenación de los datos de una cola hay multitud de métodos muy eficaces cuya descripción escapa al alcance de este libro. Sólo haremos notar que para ordenarlos no hay necesidad de intercambiar las diferentes fichas (como sucedía al ordenar tablas); basta con reajustar los punteros, lo cual supone una mayor rapidez del proceso. En la cola del primer ejemplo:



Aunque se ha introducido el concepto de puntero como algo ligado a los registros, en realidad se pueden definir punteros para apuntar a cualquier tipo de variable:

PUNT: ^ARRAY[1..10000] OF INTEGER;

pero sólo en casos especiales tienen interés (como, por ejemplo, con el compilador TURBO PASCAL, para tener más de 64 Kbytes de variables «seudo-estáticas»).



Control de la memoria ocupada

En el programa COLA se podrían introducir fichas indefinidamente hasta llegar a llenar toda la memoria disponible en el ordenador. Si, al intentar reservar espacio para una nueva variable, no hubiera ya suficiente espacio, se produciría un error y, según los casos, se pararía el programa o incluso podría producirse una situación de «cuelgue» del ordenador.

Por tanto, es necesario, cada vez que se vaya a pedir nuevos datos, comprobar antes si todavía hay suficiente espacio para ellos. La mayoría de los compiladores tienen alguna función para esto. Esta función en alguna versión de PASCAL se denomina MEMAVAIL y devuelve la cantidad de memoria libre (en bytes o en algún otro tipo de medida) en el momento de la llamada:

IF MEMAVAIL < MINIMO THEN WRITELN ('Se acabó la memoria.');

La utilización de almacenamiento dinámico elimina la obligación de hacer previsiones sobre el número de datos que se van a procesar, y los punteros permiten, además, crear fácilmente estructuras de tipo cola o árbol (de la que veremos un ejemplo). Para completar el panorama sólo hace falta tener la posibilidad de, opcionalmente, dejar libre la zona de memoria de las variables dinámicas que no se vayan a utilizar más para así poder reservar otras nuevas.

Esto se consigue en PASCAL mediante los procedimientos predefinidos MARK y RELEASE. Si escribimos:

MARK (P);

donde P es un puntero de cualquier tipo, al ejecutarse el procedimiento se guarda en P la posición de la última zona de memoria ocupada. Posteriormente, si se llega a la instrucción:

RELEASE (P);

toda la memoria que se reservó para variables dinámicas tras ejecutarse **MARK (P)** queda libre para otras nuevas. Los datos de aquellas variables se pierden y, por tanto, hay que tener mucho cuidado al utilizar estos procedimientos.

Algunos compiladores disponen también del procedimiento **DISPOSE**. Con él es posible dejar libre sólo la zona que corresponda a una variable específica. Si ponemos:

DISPOSE (ULTIMO);

la zona ocupada por la variable a que apunta **ULTIMO** queda libre, o sea, es una especie de «anti-NEW». Utilizando este método la memoria se libera a trozos, por lo que es posible que al cabo del tiempo, aunque el total disponible sea grande, esté muy fragmentado.

Por ello, suele haber, además de **MEMAVAIL** o su equivalente, otra función para conocer el tamaño del mayor pedazo disponible (en algún caso denominada **MAXAVAIL**), pues aunque hubiera suficiente memoria libre para un registro, pudiera ser que no hubiera ningún pedazo lo suficientemente amplio para albergarlo.

En cualquier caso, el sistema **MARK / RELEASE** no se debe utilizar junto con **DISPOSE** a la hora de liberar memoria.



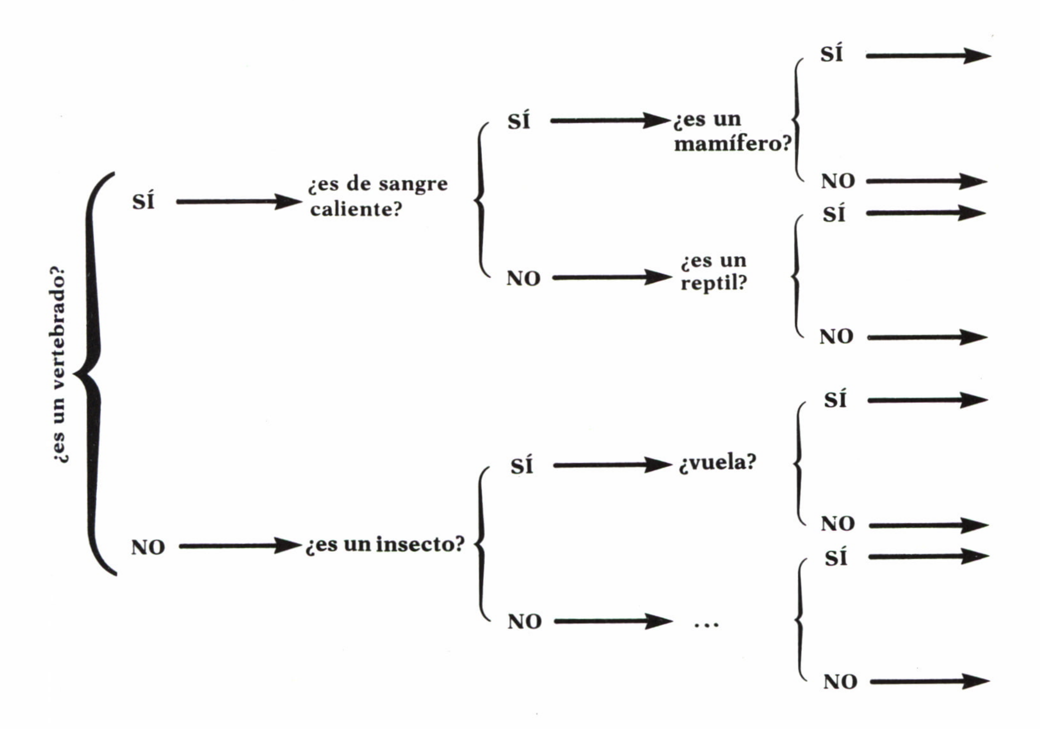
Un programa de ejemplo

Como último ejemplo de almacenamiento dinámico de registros, vamos a escribir un programa capaz de hacer preguntas para intentar adivinar un animal y, caso de no poder hacerlo, tomar nota del animal que era y de sus características para tenerlo en cuenta en sucesivas ocasiones.

La primera vez el programa preguntará directamente por un animal en concreto, por ejemplo, la mosca. Si acertara, ahí acabaría todo. Sin embargo, imaginemos que el animal que habíamos pensado fuera el perro. Al contestar al programa que no era la mosca, éste nos preguntaría qué animal era el escogido (el perro) y alguna propiedad que lo distinguiera de la mosca (ladra). Con esta nueva información, al siguiente intento de descubrir un animal, el programa preguntaría primero si ladra, para, según fuera la respuesta, preguntar a continuación por el perro o por la mosca. De esta manera, el ordenador iría «aprendiendo» nuevos animales y nuevas preguntas que hacer antes de preguntar por uno en concreto.

Supongamos que, en un momento dado, la primera pregunta de todas fuese: «¿es un vertebrado?». Si la respuesta a ella fuese **SÍ**, habría que hacer a continuación alguna pregunta lógica para vertebrados, por ejemplo: «¿es de sangre caliente?», mientras que si hubiera sido **NO**, la siguiente pre-

gunta debería ser otra distinta. Para saber la secuencia de preguntas a hacer podríamos utilizar un esquema como el siguiente:



y así hasta llegar a preguntar por un animal en concreto.

A la estructura que resulta se le denomina ESTRUCTURA TIPO ARBOL. En una estructura semejante, a cada punto de bifurcación se le llama NODO y al primer nodo de todos, RAIZ del árbol.

Crear estructuras de datos de tipo árbol en PASCAL es muy fácil. En el caso que nos ocupa bastaría con utilizar una variable de tipo registro para cada nodo. Estos registros deberían tener un campo para guardar la pregunta y otros dos de tipo puntero para indicar a qué registros se debe acudir según la respuesta. Tras los nodos que preguntan por animales concretos no habría más registros.

El programa empezaría así:

```
PROGRAM ANIMALES;

CONST
 LONGPREG = 50;  (* máximo número de caracteres por pregunta *)
```

```
TYPE
 TIPOPREG = ARRAY [1..LONGPREG] OF CHAR; (* para las preguntas *)
 PUNTERO  = ^TIPOFICHA;
 TIPOFICHA= RECORD
             PREGUNTA: TIPOPREG;
             QUESI,QUENO: PUNTERO
            END;
```

Veamos ahora el procedimiento para determinar la secuencia de preguntas a partir de un nodo dado:

«Mirar en árbol desde el nodo tal:»

— ¿Es un nodo final?

SÍ: Preguntar por el animal que contiene.

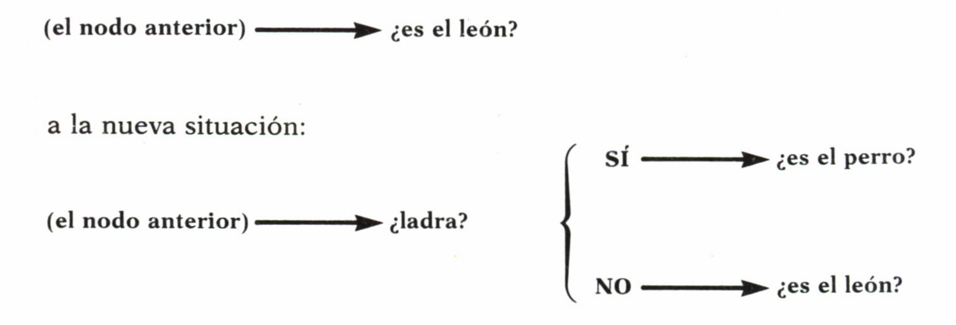
- — Si se ha acertado, se acabó.
- — Si se ha fallado, preguntar en qué animal se ha pensado y sus propiedades, ampliar con ello el árbol y terminar.

NO: Formular la pregunta que contiene.

- — Si la respuesta es SÍ, mirar en árbol desde el nodo indicado por el puntero QUESI.
- — Si la respuesta es NO, mirar en árbol desde el nodo indicado por el puntero QUENO.

Como se ve, el procedimiento es recursivo, pues se llama a sí mismo. Para empezar una búsqueda se llamaría al procedimiento desde el programa principal para «mirar en árbol desde el nodo raíz».

Veamos ahora cómo ampliar el árbol con nueva información. Supongamos que se ha llegado a un nodo final que contiene «el león» y que la respuesta ha sido NO. Si el animal resultara ser «el perro» y la propiedad que lo distingue «ladra», deberíamos pasar de la situación:



o sea, antes de preguntar por el león, preguntar si ladra por si acaso es el perro. El procedimiento sería:

«Ampliar el árbol en el nodo tal:»

1. Reservar sitio para dos nuevos registros a los que apunten QUESI y QUENO.
2. Guardar en el primero de ellos el nuevo animal y hacer sus dos punteros iguales a NIL (es un nodo final).
3. Guardar en el segundo el animal por el que se preguntó y hacer sus dos punteros iguales a NIL.
4. Guardar la propiedad distintiva en el nodo en cuestión.

El programa definitivo quedaría:

```
PROGRAM ANIMALES;

CONST
 LONGPREG = 50;  (* máximo número de caracteres por pregunta *)
 (* si se cambia, retocar también el principio del programa  *)
TYPE
 TIPOPREG = ARRAY [1..LONGPREG] OF CHAR;
 PUNTERO  = ^TIPOFICHA;
 TIPOFICHA= RECORD
             PREGUNTA: TIPOPREG;
             QUESI,QUENO: PUNTERO
            END;
VAR
 PRIMERO: ^TIPOFICHA; (* Sirve para apuntar al nodo raíz *)
 C: CHAR;
(*-----------------------------------------------------------*)
PROCEDURE PONPREGUNTA (T: TIPOPREG);
 (* Escribe la pregunta quitando lo que sobra por la *)
 (* derecha para que la interrogación quede pegada   *)
 VAR I,J: INTEGER;
BEGIN
 (* retroceder desde la derecha hasta encontrar la última letra: *)
 I:=LONGPREG;
 WHILE (T[I] = ' ') OR (T[I] = CHR(0)) DO  I:=I-1;
 (* luego escribir el texto hasta ella: *)
 FOR J:=1 TO I DO WRITE (T[J]);
 WRITE ('? (S/N) ')
END;
(*-----------------------------------------------------------*)
FUNCTION  AFIRMATIVO: BOOLEAN;
 (* Lee respuesta y devuelve TRUE si ha sido SI *)
 VAR C: CHAR;
```

```
BEGIN   READLN (C); AFIRMATIVO := (C<>'n') AND (C<>'N')   END;
(*----------------------------------------------------------------*)
PROCEDURE AMPLIAR (Q: PUNTERO);
(* Amplia el árbol en el nodo apuntado por Q *)
 VAR COPIA: TIPOFICHA;
BEGIN
 WITH Q^ DO
  BEGIN
   NEW (QUESI);
   NEW (QUENO);

   WRITELN;
   WRITE ('Que animal es ? (con "el" o "la" por delante): ');
   READLN (QUESI^.PREGUNTA);
   QUESI^.QUESI:=NIL;
   QUESI^.QUENO:=NIL;

   QUENO^.PREGUNTA:=PREGUNTA;
   QUENO^.QUESI:=NIL;
   QUENO^.QUENO:=NIL;
   REPEAT
    WRITELN ('Déme una propiedad que lo distinga:');
    READLN (PREGUNTA);
    WRITELN ('Entonces, si pregunto:');
    PONPREGUNTA (PREGUNTA);
    WRITELN;
    WRITE ('y la respuesta es SI, puedo suponer que es ');
    PONPREGUNTA (QUESI^.PREGUNTA); WRITELN;
   UNTIL AFIRMATIVO;

   WRITELN;
   WRITELN ('Gracias. Hasta otra.')
  END
END;

(*----------------------------------------------------------------*)
PROCEDURE MIRAAVEREN (P: PUNTERO);
(* Avanza por el árbol desde el nodo apuntado por P *)
 VAR C: CHAR;
BEGIN
 WITH P^ DO
  IF (QUESI=NIL) AND (QUENO=NIL) THEN  (* si es final: *)
    BEGIN
     WRITE ('Es ');
     PONPREGUNTA (PREGUNTA);
     IF  NOT AFIRMATIVO  THEN  AMPLIAR (P)
           ELSE BEGIN WRITELN; WRITELN ('VALE, HASTA OTRA') END;
    END
```

```
      ELSE
        BEGIN
         PONPREGUNTA (PREGUNTA);
         IF AFIRMATIVO THEN MIRAAVEREN (QUESI)
                       ELSE MIRAAVEREN (QUENO)
        END
END;
(*------------------------------------------------------------*)

BEGIN
(* Al principio sólo hay un nodo que por fuerza ha de ser final:*)
 NEW (PRIMERO);
 WITH PRIMERO^ DO
  BEGIN
   PREGUNTA:='la mosca
  (* Por poner algo. Ya irá aprendiendo. *)
   QUESI:=NIL;
   QUENO:=NIL
  END;

 REPEAT
  WRITELN;
  WRITELN ('Piense en un animal y pulse INTRO para seguir.');
  READLN (C); (* leer algo para esperar el Intro *)
  MIRAAVEREN (PRIMERO);
  WRITE ('Desea seguir (S/N) ? ');
 UNTIL NOT AFIRMATIVO

END.
```

Gracias al empleo de almacenamiento dinámico, el árbol puede crecer con el único límite de la memoria que haya disponible.

Aunque como ejemplo de programa de inteligencia artificial deja bastante que desear (no hay control sobre el crecimiento equilibrado del árbol, las preguntas no están jerarquizadas, es imposible modificar los nodos existentes y, sobre todo, cuando se acaba el programa se pierden todos los datos trabajosamente tecleados), es un buen ejemplo de utilización de árboles, de lo fácil que resulta con PASCAL y de cómo los procedimientos recursivos resultan ser muchas veces la solución más sencilla.

Nota: En caso de que nuestro compilador no admitiera la definición previa del tipo puntero, haríamos:

```
               TIPOFICHA= RECORD
                            PREGUNTA:     TIPOPREG;
```

```
                QUESI,QUENO: ^TIPOFICHA
              END;
     VAR
      PRIMERO: ^TIPOFICHA;
```

Si resultara que la variable PRIMERO así definida fuese considerada como de distinto tipo que los campos QUESI y QUENO, haríamos lo siguiente:

```
VAR
 AUXILIAR: TIPOFICHA; (* Sirve para apuntar al nodo raíz *)
```

y en lugar de PRIMERO utilizaríamos AUXILIAR.QUESI. Igualmente pudiera ser que hubiera problemas al definir el tipo en la lista de parámetros de MIRAAVEREN y AMPLIAR; la solución sería pasar como parámetro un registro de tipo TIPOFICHA en cuyo campo QUESI (o QUENO) se guardaría previamente el puntero que realmente se desea transferir.

FICHEROS 13



ASTA ahora, en todos los ejemplos que utilizaban datos introducidos por teclado para su posterior proceso, éstos se perdían al acabar de ejecutarse el programa. Recordemos el programa de ordenación de fichas, o el de los animales.

Está claro que los ordenadores no habrían alcanzado la enorme difusión de nuestros tiempos si no existiera algún sistema para guardar los datos que se han introducido, o los nuevos datos obtenidos por los programas, de manera que estén disponibles para su uso en cualquier otro momento, aunque el ordenador se haya estado dedicando a otra tarea o incluso haya estado desconectado. De estas cuestiones trata este capítulo.

Existen multitud de sistemas de almacenamiento de información. De ellos el más versátil y extendido entre los ordenadores personales es, sin duda, el de disco magnético flexible o «diskette». Aunque los conceptos de este capítulo se van a tratar en plan general, están desarrollados, no obstante, con la mirada puesta en los discos flexibles.

Desafortunadamente, al ser estas cuestiones tan dependientes de cada máquina en concreto, el PASCAL estándar es poco explícito sobre ellas, por lo que hay prácticamente tantas maneras de programarlas como versiones de compilador se han desarrollado.

Por ello, se va a explicar lo que hay al respecto en el PASCAL estándar, para pasar a continuación a la confección de ejemplos con una versión concreta de compilador. Para esto se ha escogido el TURBO PASCAL, de la casa Borland, que, entre otros, funciona con los ordenadores personales IBM y compatibles; éstos son unos de los ordenadores personales dotados de disco flexible más difundidos en la actualidad y el compilador es, sin duda, el que más ha contribuido a la difusión del PASCAL y uno de los programas más vendidos en la historia de los ordenadores personales.

FICHEROS SECUENCIALES

Se denomina fichero («file», en inglés) a una secuencia de datos del mismo tipo de los que sólo uno de ellos está disponible en un momento dado. Imaginemos una cinta de magnetófono en la que se han grabado palabras una detrás de otra; el conjunto de palabras es lo que se denominaría un «fichero de palabras» y, como es lógico, en un momento dado sólo es posible escuchar una de ellas. Los ficheros de oficina son también un buen ejemplo.

Los datos que se guardan en los dispositivos de almacenamiento externo adoptan también esta estructura, de manera similar a como se encuentran las palabras guardadas en la cinta del ejemplo. Los datos se encuentran uno detrás de otro en algún tipo de soporte físico (una cinta o un disco magnético, usualmente), de manera que, en un momento dado, sólo uno de ellos se puede leer o grabar.

Al igual que en la cinta podría haber diferentes conjuntos de palabras (por ejemplo, la lista de los reyes godos, los elementos de la tabla periódica, etc.), en un dispositivo de almacenamiento puede haber diferentes ficheros, a cada uno de los cuales se le habrá dado un nombre distinto.

Cuando la única forma de llegar a un elemento específico de un fichero es comenzar por el primero, e ir recorriendo un elemento tras otro hasta llegar al deseado, se dice que el fichero es SECUENCIAL (o de acceso secuencial). Es el caso de la cinta: la única forma segura de encontrar la palabra buscada es ponerse al principio y empezar a escuchar una tras otra hasta llegar a ella.

Para cambiar o añadir información a un fichero secuencial, se hace igual que como se haría para sustituir o añadir palabras a la cinta: se va recorriendo el fichero elemento a elemento hasta llegar a la posición deseada y entonces se graba la información.

Hay una serie de programas ya preparados en los ordenadores que se encargan de manejar los dispositivos de almacenamiento. Son parte de lo que se conoce como «sistema operativo». Cuando se está trabajando con ficheros, estos programas se encargan de leer y grabar los datos en ellos cuando nuestro programa PASCAL se lo pide, quedando a su cargo el manejo de una serie de punteros propios que les indican en qué elemento de un fichero nos encontramos en un momento dado, y en qué zona concreta del medio físico (cinta, disco...) se encuentra cada fichero. En principio, no es necesario saber más sobre ellos.

En PASCAL es posible definir variables de tipo fichero secuencial, con la particularidad de que los datos que albergan no están en la memoria del ordenador, sino que corresponden a los datos de un fichero almacenado en el dispositivo externo, y de que el número de datos no está defi-

nido a priori. Lo único que se tiene en memoria en un momento dado es una copia del elemento del fichero en que nos encontramos; a su vez, es posible modificar o añadir un elemento en esa posición con datos procedentes de la memoria.

También se puede, por supuesto, avanzar al siguiente elemento o posicionarse en el primero de todos, así como inaugurar ficheros o hacerlos desaparecer.

Aunque teóricamente sería posible tener ficheros con todos sus componentes almacenados en memoria, pocos compiladores lo permiten, y además su utilidad es para casos muy especiales de proceso de datos que escapan del alcance de este libro.

El tipo de variable se describe poniendo las palabras reservadas FILE y OF seguidas del tipo de elemento que constituye el fichero. Se podría declarar previamente o describirlo al tiempo que se definen las variables. Para un fichero que contuviera los nombres de los reyes godos:

```
TYPE
 TIPOREYGODO = ARRAY [1..15] OF CHAR;
VAR
 F     : FILE OF TIPOREYGODO;
 TABLA : ARRAY [1..50] OF TIPOREYGODO;
 I     : INTEGER;
 REY   : TIPOREYGODO;
```

Para que la variable F se pueda utilizar, hace falta indicar primero al PASCAL cómo se llama el fichero en que se encuentran sus datos y en qué dispositivo se encuentra aquél, caso de existir más de uno. No hay normas sobre cómo hacer esto, por lo que depende de cada compilador.

Supongamos que esta operación ya se ha realizado; hemos dicho que en todo momento, y de manera automática, se tiene en memoria una copia del elemento del fichero sobre el que nos encontramos. Esa copia sería en el ejemplo una variable de tipo TIPOREYGODO y, como no tiene ningún identificador asociado, para referirse a ella se escribiría F ^ , que viene a significar algo como «la copia del elemento del fichero F en que nos encontramos ahora». Esta variable puede utilizarse como cualquier otra del mismo tipo, y su contenido puede modificarse con vistas a ser transferido después al fichero. Por ejemplo:

```
REY:= F^;
F^ := 'Teudiselo
```

Para operar con ficheros se tienen las siguientes funciones y procedimientos predefinidos:

EOF (F) — Esta función (End Of File, fin de fichero) devuelve el valor lógico TRUE cuando nos hemos pasado de largo y nos hemos posicionado más allá del último elemento del fichero asociado a F.

RESET (F) — Este procedimiento hace que nos coloquemos al principio del fichero con vistas a su lectura. Tras su ejecución, F^ contendría el primer elemento (a no ser que el fichero estuviera vacío, en cuyo caso su contenido sería imprevisible y EOF (F) devolvería TRUE).

GET (F) — Este otro, sin embargo, hace que se avance al elemento siguiente a aquél en que nos encontrábamos, pasando F^ a tener una copia suya. Por tanto, si no hubiera elemento siguiente, EOF (F) pasaría a valer TRUE y F^ tendría un contenido indefinido.

Con ellos, si, por ejemplo, quisiéramos leer del fichero todos los reyes que contuviera y guardarlos en la variable TABLA para su posterior utilización, podríamos hacer:

```
I:=1;           (* Empezar por el primer elemento de la tabla *)
RESET (F);                  (* Leer el fichero desde el principio *)
WHILE NOT EOF (F) DO   (* Mientras estemos sobre un elemento: *)
 BEGIN
  TABLA [I]:= F^;     (* Guardar elemento en la tabla *)
  GET (F);                      (* Pasar al siguiente *)
  I:=I+1                        (* Incrementar índice *)
 END;
```

Se ha utilizado una estructura WHILE, pues si el fichero estuviera vacío, no habría que guardar ni un dato en TABLA.

Imaginemos ahora que tenemos las notas de un examen guardadas en un fichero. Si definimos la variable NOTAS como FILE OF REAL, para calcular la nota media podríamos hacer:

```
NUMERO:= 0;
SUMA:= 0.0;
RESET (NOTAS);
```

```
WHILE NOT EOF (NOTAS) DO
 BEGIN
  SUMA:= SUMA + NOTAS^;              (* añadir la nueva nota *)
  GET (NOTAS);
  NUMERO:= NUMERO+1       (* incrementar el contador de notas *)
 END;
IF NUMERO = 0 THEN WRITELN ('Fichero vacío.')
              ELSE WRITELN ('Nota media= ',SUMA/NUMERO:6:2);
```

Para modificar o incorporar nueva información a un fichero se dispone de los siguientes procedimientos:

REWRITE (F) Este procedimiento hace que el fichero asociado a F se vacíe, perdiéndose sus datos y quedando preparado para empezar a guardar nuevos datos en él desde su comienzo. Tras su ejecución, EOF (F) pasa a valer TRUE.

PUT (F) Al ejecutarse PUT (F), el contenido de F^ se transfiere al fichero, justo en la posición en que nos encontrábamos, pasándose a continuación a la siguiente posición. Por tanto, si antes de ejecutarse estuviéramos más allá del último elemento, el nuevo quedaría a continuación, siendo ahora él el último, y quedando nuevamente posicionados más allá de éste.

Con ellas, para guardar los elementos de TABLA en el fichero asociado a F haríamos:

```
REWRITE (F);
FOR I:=1 TO TOTAL DO
  BEGIN
   F^ := TABLA [I];
   PUT (F)
  END;
```

Para añadir nuevas notas al fichero NOTAS, justo a continuación de las ya existentes, haríamos:

```
(* Nos ponemos al principio y avanzamos hasta llegar al final: *)
  RESET (NOTAS);
  WHILE NOT EOF (NOTAS) DO GET (NOTAS);
```

```
WRITELN ('Introduzca una nota negativa para acabar.');
REPEAT
 LEERNOTA (S);  (* procedimiento para leer notas desde teclado *)
 IF S>=0 THEN
   BEGIN
    NOTAS^ := S;
    PUT (NOTAS)
   END
UNTIL S<0;
```

LEERNOTA sería un procedimiento escrito por nosotros y S una variable REAL.



Ejemplo

Con el compilador escogido, para asociar un fichero de disco a una variable de tipo FILE, hay que ejecutar el procedimiento ASSIGN:

ASSIGN (F, 'GODOS. LST');   (* "asigna F a GODOS. LST" *)

El nombre del fichero podría estar especificado con una variable o expresión en lugar de con una constante entre apóstrofes. La llamada a este procedimiento ha de ser el primer paso a ejecutar para trabajar con un fichero.

Por otra parte, con este compilador, en lugar de los procedimientos GET y PUT, se utilizan los procedimientos READ y WRITE (que en este caso no son PASCAL estándar) de la siguiente manera:

READ (F, TABLA [I])

que equivale a

BEGIN TABLA [I] := F^; GET (F) END

o sea, «leer del fichero asociado a F el elemento en que estamos y guardarlo en TABLA [I], para después posicionarse en el siguiente». Análogamente:

WRITE (NOTAS,S)   (*Guarda en el fichero NOTAS la variable S *)

que equivale a

BEGIN NOTAS^:= S; PUT (NOTAS) END

En el primer parámetro de la lista debe ser de tipo FILE, mientras que el segundo debe ser una variable del tipo constitutivo del fichero.

Como observará el lector, los ejemplos anteriores de lectura y escritura en ficheros se ven simplificados salvo en el caso de la búsqueda del fin de fichero, en que para avanzar habría que utilizar READ con una variable REAL cualquiera en lugar de GET:

```
WHILE NOT EOF (NOTAS) DO READ (NOTAS,R);
```

Cuando ya se ha terminado de trabajar con un fichero, hay que indicárselo al sistema operativo para que, en su caso, haga efectivos los cambios introducidos por medio del procedimiento CLOSE:

CLOSE (F)   (* "cierra el fichero asociado a F" *)

Explicadas sus peculiaridades, podemos comenzar ya con el ejemplo. Consiste en modificar el programa de los animales para poder salvar los datos en disco.

Para guardar un árbol, hay que salvar uno por uno, secuencialmente, todos sus nodos. Veamos un procedimiento para salvar un nodo dado junto con los pertenecientes a las ramas que de él salen:

«Salvar el nodo Tal y los que de él dependen:»

1. Salvar el nodo en cuestión.
2. Si no es nodo final, entonces:
   — Salvar el nodo al que apunta QUESI y los que de él dependen.
   — Salvar el nodo al que apunta QUENO y los que de él dependen.

Para salvar el árbol, bastaría con «salvar el nodo raíz y los que de él dependen». No hace falta decir que el procedimiento es recursivo. Intente el lector imaginar cómo sería un procedimiento no recursivo con idéntico cometido; aunque es factible, elaborarlo no es tarea fácil.

Esta manera de recorrer el árbol se denomina «preorden»; si el nodo en cuestión se salvará después que sus ramas, sería un «post-orden», y si se hiciera entre ambas ramas, sería un «orden central». Cualquier método sería válido en nuestro caso.

Si la variable de tipo FILE se llamara FICHERO, el procedimiento quedaría:

```
PROCEDURE SALVAR (P: PUNTERO);
(* Guarda en disco el nodo al que apunta P, y sus ramas *)
 BEGIN
```

```
WRITE (FICHERO;P^); (* Salvar ficha apuntada por P y avanzar *)
WITH P^ DO
 BEGIN
  IF QUESI <> NIL THEN SALVAR (QUESI);
  IF QUENO <> NIL THEN SALVAR (QUENO)
 END
END;
```

Para salvar el árbol se ejecutaría SALVAR (PRIMERO). Por supuesto, antes hay que llamar a los procedimientos ASSIGN y REWRITE, y después al procedimiento CLOSE.

Para recuperar un árbol almacenado en disco, se iría leyendo el fichero secuencialmente y reconstruyendo el árbol nodo a nodo.

Los campos QUESI y QUENO de cada nodo contienen las direcciones en memoria que tenían los siguientes nodos cuando se salvó el árbol. Estas no tienen por qué ser iguales al reconstruirlo en un momento posterior (puede ser un programa distinto, con más variables que ocupen sitio en memoria y desplacen la posición de ciertos datos...). Por tanto, habrá que cambiar esos campos tras leer cada nodo del disco.

Sin embargo, su contenido sí es útil. Si los campos QUESI y QUENO de un nodo son distintos de NIL, eso quiere decir que, cuando se ejecutó el procedimiento SALVAR, tras el nodo se salvaron las ramas que de él salían, que, por tanto, se encuentran a continuación en el fichero. El procedimiento ha de reflejar, así, el recorrido en preorden:

«Recoger el nodo Tal y los que de él dependen:»

1. Reservar sitio en memoria para el nodo, dejando listo el puntero que llevará hasta él.
2. Recuperar el nodo de disco.
3. Si no es nodo final, entonces:

   — Recoger el nodo al que apunta QUESI y los que de él dependen.
   — Recoger el nodo al que apunta QUENO y los que de él dependen.

En definitiva:

```
PROCEDURE RECOGER (VAR P: PUNTERO);
(* Recoge de disco el nodo al que apuntará P, y sus ramas *)
 BEGIN
  NEW (P);
  READ (FICHERO,P^);
  WITH P^ DO
   BEGIN
    IF QUESI <> NIL THEN  RECOGER (QUESI);
    IF QUENO <> NIL THEN  RECOGER (QUENO)
   END
 END;
```

Como el dato que se pasa es el puntero que lleva al nodo, el paso de parámetro ha de ser por nombre para que, tras ejecutarse NEW, se quede realmente apuntando a la zona de memoria reservada.

Tras ejecutarse ASSIGN y RESET para posicionarnos al principio del fichero, se ejecutaría RECOGER (PRIMERO).

Por otra parte, podríamos definir la constante NOMBRE, para el nombre del fichero:

```
CONST
 NOMBRE = 'ANIMALES.ARB';
 (* ... las otras constantes que pudiera haber ... *)
```

y la variable FICHERO:

```
VAR
 FICHERO: FILE OF TIPOFICHA;
 (* ... las otras variables ... *)
```

Con todo esto, sólo quedaría modificar el programa principal:

```
BEGIN

WRITE ('Leo datos de disco (S/N) ? ');
IF AFIRMATIVO THEN
  BEGIN
   ASSIGN (FICHERO, NOMBRE);
   RESET (FICHERO);
   RECOGER (PRIMERO);
   CLOSE (FICHERO)

ELSE
 BEGIN
  NEW (PRIMERO);
  WITH PRIMERO^ DO
   BEGIN
    PREGUNTA:='la mosca
    QUESI:=NIL;
    QUENO:=NIL
   END
 END;
```

```
REPEAT
  WRITELN;
  WRITELN ('Piense en un animal y pulse INTRO para seguir.');
  READLN (C);
  MIRAAVEREN (PRIMERO);
  WRITE ('Desea seguir (S/N) ? ');
UNTIL NOT AFIRMATIVO;

WRITE ('Guardo los datos (S/N) ? (reemplazando al anterior árbol) ');
IF AFIRMATIVO THEN
  BEGIN
   ASSIGN (FICHERO, NOMBRE);
   REWRITE (FICHERO);
   SALVAR (PRIMERO);
   CLOSE (FICHERO)
  END

END.
```

Cuando se ejecuta REWRITE de un fichero que todavía no existe, se crea uno con ese nombre que queda listo para albergar datos.

El programa resultante no dispone de tratamiento de errores, por lo que si, al intentar leerlo, no existiera el fichero ANIMALES.ARB, o, al intentar salvar los datos, no hubiera suficiente espacio en disco o éste estuviera protegido contra grabación, se produciría un error y se detendría el programa. La manera de evitar esto debe buscarla el lector en el Manual del compilador correspondiente.



FICHEROS DE TEXTO

Muy frecuentemente lo que se desea guardar en un fichero son datos tal como aparecerían en la pantalla del ordenador, o sea, frases, palabras o números representados con los caracteres que les corresponden. Se suele guardar así información que en algún momento ha de ser leída tal cual, sin ser sometida a ningún proceso. Una carta, un capítulo de este libro o el texto de un programa PASCAL son casos típicos.

Para ello habría que utilizar estructuras de tipo FILE OF CHAR que permitirían guardar los datos en forma de chorro de caracteres.

Los textos normalmente se encuentran divididos en renglones, siendo lo más corriente al guardarlos en ficheros utilizar unos caracteres especiales para indicar cuándo acaba un renglón y empieza el siguiente. Con los códigos ASCII el fin de una línea se indica a menudo con la pareja formada por los caracteres cuyos ordinales son 13 y 10.

Volviendo al ejemplo de la cinta, es como si, habiendo grabado un li-

bro en ella, se hubiera indicado el final de cada renglón con un silbido o una palmada.

Tan habituales son estos ficheros que su tipo se encuentra ya predefinido con el nombre TEXT.

Como tales ficheros de caracteres, se podrían utilizar todos los procedimientos vistos hasta ahora para leer o escribir de carácter en carácter; sin embargo, el PASCAL permite hacer un tratamiento un poco especial de los ficheros de tipo TEXT para manejar cómodamente la cuestión de las líneas, por medio de los procedimientos READ, READLN, WRITE y WRITELN junto con la función EOLN. Si definimos:

```
VAR T: TEXT;      (* más o menos como poner T: FILE OF CHAR *)
```

T sería una variable con un fichero de texto asociado, y podríamos hacer lo siguiente con ella:

EOLN (T)	Esta función (End Of LiNe, fin de línea) devuelve TRUE si nos encontramos al final de una línea, y FALSE en caso contrario.
READLN (T)	Cuando se está leyendo el fichero asociado a T, con esto nos posicionaríamos en el primer carácter de la siguiente línea a aquélla en que nos encontramos.
WRITELN (T)	Al escribir en el fichero asociado, esto pondría la marca de fin de línea justo a continuación del último carácter escrito, quedando todo listo para comenzar a escribir los de la siguiente línea.

Por otra parte, con los ficheros de texto SI está definida en el PASCAL estándar la utilización de WRITE y READ que hemos empleado en el último ejemplo:

```
READ  (T,C)   equivale a   BEGIN C:=T^, GET (T) END
WRITE (T,C)   equivale a   BEGIN T^:=C; PUT (T) END
```

o sea, leer (o escribir) el carácter C en la posición del fichero en que nos encontramos, para pasar después a la siguiente posición.

Sin embargo, la utilización de READ y WRITE va más allá: es posible leer o escribir varios datos con una sola instrucción:

```
READ (T,C1,C2)   equivale a   READ (T,C1); READ (T,C2)
```

y análogamente con WRITE. Si tras una instrucción READ o WRITE, respectivamente, hubiera que ejecutar READLN o WRITELN, se podría utilizar una sola instrucción:

```
READLN (T,C1,C2)   equivale a   READ (T,C1); READ (T,C2); READLN (T)
WRITELN (T,C)      equivale a   WRITE (T,C); WRITELN (T)
```

Esto último, por ejemplo, supondría escribir el carácter C seguido de una marca de fin de línea.

Además, no sólo se pueden leer o escribir caracteres; es posible escribir valores de tipo INTEGER o REAL utilizando incluso definiciones de espaciado, de manera que lo que se escriba sea su representación con caracteres. Si, por ejemplo, pusiéramos:

```
WRITE (T,N:3)
```

y N fuera una variable INTEGER con valor 54, esto sería equivalente a poner:

```
WRITE (T,' '); WRITE (T,'5'); WRITE (T,'4')
```

También se pueden escribir en un fichero valores de tipo BOOLEAN, lo que equivaldría a escribir la palabra 'FALSE' o 'TRUE', y, normalmente, strings y datos de tipo ARRAY de caracteres.

Igualmente, si el fichero contuviera, por ejemplo, los cuatro caracteres ' 36 ' y estuviéramos posicionados sobre el primero de ellos (un espacio en blanco) o el segundo, la ejecución de

```
READ (T,N)     o bien     READLN (T,N)
```

haría que la variable entera N tomara el valor 36, quedando después posicionados en el carácter siguiente al seis o al comienzo de la siguiente línea, según el caso.

En otras palabras, con READ, READLN, WRITE y WRITELN se puede hacer EXACTAMENTE lo mismo que hacíamos para escribir datos en pantalla o leerlos de teclado, solo que enviando los caracteres que normalmente aparecerían en la pantalla a un fichero, y recogiendo los datos que normalmente se leerían del teclado de un fichero. Al leer datos de un fichero, la marca de fin de línea sería equivalente a pulsar la tecla de RETURN (intro, newline, etc.) cuando se lee del teclado.

Esta coincidencia no es casual. La presentación en pantalla toma la estructura de una secuencia de caracteres distribuida en renglones, y los datos que se introducen desde teclado son también secuencias de caracteres separadas entre sí a golpes de RETURN. Por ello, en PASCAL se trata al dispositivo de presentación de datos (normalmente la pantalla) y al de recogida de datos (normalmente el teclado) como si fueran ficheros. Unos ficheros un poco especiales, pues sus datos tienen una vida efímera. Por ejemplo, una vez que se han leído datos de teclado, no es posible volver a

posicionarse en el primero de ellos para repetir la lectura, y mientras en uno de ellos sólo se puede escribir, con el otro sólo se puede leer.

Estos ficheros se encuentran predefinidos y se llaman, respectivamente, OUTPUT (salida, en inglés) e INPUT (entrada). Los procedimientos RESET y REWRITE no se utilizan, como es lógico, con estos ficheros.

Para escribir un dato en el fichero OUTPUT y leer otro del fichero INPUT, haríamos:

```
WRITE (OUTPUT,N);
READ  (INPUT, A);
```

Cuando en un procedimiento de los existentes para ficheros no aparece como primer parámetro el nombre de una variable de tipo FILE, el PASCAL supone que el fichero que falta es, según el caso, OUTPUT o INPUT, por lo que lo anterior equivale a:

```
WRITE (N);
READ (A)
```

En otras palabras, todos los ejemplos del libro en que hemos utilizado esos procedimientos han sido casos particulares de escritura o lectura de ficheros de tipo TEXT. En todos esos ejemplos podríamos, por tanto, haber enviado datos a un fichero en lugar de a la pantalla para su posterior utilización, o leído los datos de un fichero en lugar del teclado, sin más que colocar en cada instrucción de entrada o salida el nombre de la variable de tipo FILE asociada al fichero (y hacer los preparativos previos para manejar ficheros que ya conocemos).

Para enviar datos a otros dispositivos, el método suele ser el mismo. Por ejemplo, es corriente que, caso de haber impresora, ésta figure como un fichero con un nombre predefinido. Este fichero sería similar al fichero OUTPUT.

La posibilidad de salvar y recoger números en ficheros utilizando no los códigos que internamente emplea el ordenador, sino su representación por medio de caracteres, tal como los escribiría un ser humano, sirve para que datos preparados y salvados por un programa en un ordenador dado se puedan leer con otro programa preparado con un compilador distinto o con un ordenador de distinto tipo.



Ejemplos

Con el compilador escogido no hay excepciones importantes a lo que hemos descrito.

En primer lugar, vamos a escribir un programa para guardar en un fichero de texto los datos que tecleemos. Los datos tecleados para cada línea se recogerán en una variable **ARRAY OF CHAR**, pero en el fichero sólo se guardarán los caracteres realmente tecleados. (Cuando el número de éstos es menor que los que tiene definidos la variable, al leerse ésta se completa automáticamente con espacios en blanco.)

```
PROGRAM GUARDAR;

TYPE
 TIPORENGLON = ARRAY [1..80] OF CHAR;
VAR
 FICHERO: TEXT;
 NOMBRE : ARRAY [1..20] OF CHAR; (* Para el nombre del fichero *)
 RENGLON: TIPORENGLON;       (* para leer de teclado cada línea *)
 I,LONGITUD : INTEGER;

BEGIN
 WRITE ('Nombre del fichero: '); READLN (NOMBRE);
 ASSIGN (FICHERO,NOMBRE);
 REWRITE (FICHERO);
 WRITELN ('Empiece a escribir el texto, separando las líneas ',
          'con INTRO.');
 WRITELN ('Para acabar, pulse INTRO dos veces.');
 REPEAT
  READLN (RENGLON);
 (* retroceder desde el final hasta primer carácter no blanco *)
  LONGITUD:= 80;
  WHILE (RENGLON [LONGITUD]=' ') AND (LONGITUD > 1) DO
    LONGITUD:=LONGITUD-1;
  IF RENGLON [LONGITUD]=' ' THEN LONGITUD:=0;

  IF  LONGITUD > 0 THEN   (* si hay caracteres *)
   BEGIN
    FOR I:=1 TO LONGITUD DO WRITE (FICHERO, RENGLON [I]);
    WRITELN (FICHERO)      (* tras cada línea, marcar el final *)
   END
 UNTIL LONGITUD <= 0;

 CLOSE (FICHERO)
END.
```

Aunque pudiera parecer que la eliminación de blancos se puede programar de manera más sencilla, se ha hecho así para evitar que se pueda llegar a hacer referencia a RENGLON [0] al pulsarse Intro dos veces seguidas, pues entonces RENGLON sería todo espacios en blanco.

Este programa serviría, por ejemplo, para escribir programas PASCAL línea a línea y guardarlos en un fichero. A diferencia de un buen programa editor, no es posible corregir los errores una vez se ha pasado a la siguiente línea.

Por último, vamos a escribir un programa para presentar por pantalla el contenido de un fichero de texto:

```
PROGRAM RECOGER;

VAR
 NOMBRE : ARRAY [1..20] OF CHAR; (* Para el nombre del fichero *)
 FICHERO: TEXT;
 C      : CHAR;

BEGIN
 WRITE ('Nombre del fichero: '); READLN (NOMBRE);
 ASSIGN (FICHERO,NOMBRE);
 RESET (FICHERO);
 CLRSCR;

 WHILE NOT EOF (FICHERO) DO
 (* mientras quede algo, presentar líneas: *)
  BEGIN
    WHILE NOT EOLN (FICHERO) DO
     (* mientras queden caracteres en la línea actual: *)
      BEGIN
       READ (FICHERO,C);     (* leer carácter *)
       WRITE (C)             (* presentarlo   *)
      END;
    READLN (FICHERO);  (* Tras cada línea, pasar a la siguiente *)
    WRITELN
  END;

 CLOSE (FICHERO)
END.
```

ACCESO ALEATORIO A FICHEROS

Aunque no está definido en el PASCAL estándar, casi todos los compiladores que permiten el manejo de ficheros contemplan este tipo de acceso.

El acceso aleatorio consiste, a diferencia del acceso secuencial, en poderse posicionar en el elemento que se quiera de un fichero sin necesidad de comenzar por el primero e ir recorriendo todos uno detrás de otro. Esto es posible cuando los elementos que componen el fichero ocupan todos el mismo espacio de almacenamiento, pues entonces es posible calcular a qué «distancia», por decirlo de alguna manera, se encuentra el deseado. En el ejemplo de la cinta, si todas las palabras ocuparan la misma longitud de cinta, sería posible saber cuánto hay que rebobinar o avanzar para posicionarse sobre otra dada.

Para ello, suele existir un procedimiento que podría ser algo como:

SEEK (F,N)   (* "Seek" es buscar en inglés *)

donde F sería una variable de tipo FILE y N algún valor INTEGER.

Si N valiera, por ejemplo, 17, tras su ejecución quedaríamos posicionados sobre el elemento decimoséptimo del fichero asociado a F. De esta manera, podría utilizarse un fichero de manera similar a como se utiliza una variable de tipo ARRAY.

Además, suele existir alguna función para obtener el número de elementos que tiene un fichero.

Cuando el fichero es de caracteres, lo único que se podría hacer es posicionarse sobre el carácter número tal, pero no sobre la línea número cuál, pues el tamaño de éstas no es constante.

Se podría imitar el procedimiento SEEK para un tipo de fichero específico, a base de colocarse al principio del fichero cada vez y recorrer lo que se necesite, pero, como imaginará el lector, puede ser terriblemente lento:

```
TYPE TIPOFICHERO = FILE OF TIPOELEMENTO;

PROCEDURE SEEK (F: TIPOELEMENTO, N: INTEGER);
 VAR I: INTEGER;
BEGIN  RESET (F);  FOR I:=1 TO N DO GET (F) END;
```

BIBLIOGRAFIA

Algoritmos + Estructuras de datos = Programas. Niklaus Wirth. Ediciones del Castillo.

Programación estructurada. O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare. Tiempo Contemporáneo.

PASCAL. User Manual and Report. Kathleen Jensen & Niklaus Wirth. Springer-Verlag.

Recursive techniques in programming. D.W. Barron. Macdonald / Elsevier.

Scientific PASCAL. Harley Flanders. Reston Publishing Co. Inc.

Programming in MODULA-2. Niklaus Wirth. Springer-Verlag.

ENCICLOPEDIA PRACTICA DE LA

INFORMATICA APLICADA

INDICE GENERAL

5 AÑADA PERIFERICOS A SU ORDENADOR

Breve descripción de varios periféricos que facilitan la comunicación con el ordenador personal, con algunos ejemplos de fácil construcción: ratón, lápiz óptico, marco para pantalla táctil...

6 PRACTIQUE MATEMATICAS Y ESTADISTICA CON EL ORDENADOR

En este libro se repasan los principales conceptos de las Matemáticas y la Estadística, desde un punto de vista eminentemente práctico y para su aplicación al ordenador personal. Se basan los diferentes textos en la presentación de pequeños programas (que usted podrá introducir en su ordenador personal).

7 APL: LENGUAJE PARA PROGRAMADORES DIFERENTES

APL es un lenguaje muy potente que proporciona gran simplicidad en el desarrollo de programas y al mismo tiempo permite programar sin necesidad de conocer todos los elementos del lenguaje. Por ello es ideal para quienes reúnan imaginación y escasa formación en Informática.

8 DISPOSITIVOS INTERACTIVOS PARA SU ORDENADOR

Descripción detallada de la forma de construir, paso a paso y en su propia casa, dispositivos electrónicos que aumentarán la potencia y facilidad de uso de su ordenador: tableta digitalizadora, convertidores de señales analógicas, comunicaciones entre ordenadores.

9 CRIPTOGRAFIA: LA OCULTACION DE MENSAJES Y EL ORDENADOR

En este libro se presentan las técnicas de ocultacion de mensajes a través de la criptografía desde los primeros tiempos hasta la actualidad, en que el uso de los computadores ha proporcionado la herramienta necesaria para llegar al desarrollo de esta ciencia.

10 PRACTIQUE CIENCIAS NATURALES CON EL ORDENADOR

Ejemplos sencillos para practicar con el ordenador. Casos curiosos de la Naturaleza en forma de programas para su ordenador personal.

11 GRAFICOS ANIMADOS CON EL ORDENADOR

En este libro las técnicas utilizadas para la animación son el resultado de unas pocas ideas básicas muy sencillas de comprender. Descubrirá los trucos y secretos de movimientos, choques, rebotes, explosiones, disparos, saltos, etc.

12 JUEGOS INTELIGENTES EN MICROORDENADORES

Los ordenadores pueden enfrentarse de forma «inteligente» ante puzzles y otros tipos de juegos. Esto es posible gracias al nuevo enfoque que ha dado la IA a la tradicional teoría de juegos.

13 ECONOMIA DOMESTICA CON EL ORDENADOR PERSONAL

Breve introducción a la contabilidad de doble partida y su aplicación al hogar, con explicaciones de cómo utilizar el ordenador personal para facilitar los cálculos, mediante un programa especialmente diseñado para ello.

14 COMO SIMULAR CIRCUITOS ELECTRONICOS EN EL ORDENADOR

Introducción a los diferentes métodos que se pueden emplear para simular y analizar circuitos electrónicos, mediante la utilización de diferentes lenguajes.

15 LOS LENGUAJES DE LA INTELIGENCIA ARTIFICIAL

Libro en que se describen los lenguajes específicos para la «elaboración del saber» y los entornos de programación correspondientes. El conocimiento de estos lenguajes, además de interesante en sí mismo, es sumamente útil para entender todo lo que la Informática Artificial supondrá para el futuro de la Informática.

16 PRACTIQUE FISICA Y QUIMICA CON SU ORDENADOR

Libro eminentemente práctico para realizar pequeños «experimentos» con su ordenador y distraerse de un modo útil.

17 EL ORDENADOR Y LA LITERATURA

En este libro se examinan procesadores de textos, programas de análisis literario y una curiosa aplicación desarrollada por el autor: APOLO, un programa que compone estructuras poéticas.

18 COMO ELEGIR UNA HOJA ELECTRONICA DE CALCULO

En este título se estudian las diferentes versiones existentes de esta aplicación típica, desde el punto de vista de su utilidad para, en función de las necesidades de cada usuario y del ordenador de que dispone, poder elegir aquella que más se adecúe a cada caso.

19 DIBUJOS TRIDIMENSIONALES EN EL ORDENADOR PERSONAL

Compruebe que también con su ordenador personal puede llegar a diseñar y calcular imágenes en tres dimensiones con técnicas semejantes a las utilizadas por los profesionales del dibujo con equipos mucho más sofisticados.

20 ¿MAQUINAS MAS EXPERTAS QUE LOS HOMBRES?

Después de situar los «sistemas expertos» en el contexto de la inteligencia artificial y describir su construcción, su funcionamiento, su utilidad, etc., se analiza el papel que pueden tener en el futuro (y presente, ya) de la Informática.

21 PRACTIQUE HISTORIA Y GEOGRAFIA CON SU ORDENADOR
Libro interesante para los aficionados a estas ciencias, a quienes presenta una nueva visión de cómo utilizar el microordenador en su estudio.

22 ERGONOMIA: COMUNICACION EFICIENTE HOMBRE-MAQUINA
Análisis de la comunicación entre el hombre y la máquina, y estudio de diferentes soluciones que tienden a facilitarla lo más posible.

23 EL ORDENADOR Y LA ASTRONOMIA
Los cálculos astronómicos y el conocimiento del firmamento en un libro apasionante y curioso.

24 VISION ARTIFICIAL. TRATAMIENTO DE IMAGENES POR ORDENADOR
El procesado de imágenes es un campo de reciente y rápido desarrollo con importantes aplicaciones en áreas tan diversas como la mejora de imágenes biomédicas, robóticas, teledetección y otras aplicaciones industriales y militares. Se presentan los principios básicos, los sistemas y las técnicas de procesado más usuales.

25 LA ESTACION TERMINAL PERSONAL
Las modernas técnicas de comunicación van permitiendo que las grandes capacidades de proceso y el acceso a bases de datos de gran tamaño estén cada día más al alcance de cada usuario (fuera ya de los Centros de Proceso de Datos).

26 EL ORDENADOR COMO MAQUINA DE ESCRIBIR INTELIGENTE
Descripción de los sistemas de tratamiento de textos existentes, análisis comparativos y estudio de posibilidades de cada uno de ellos. Guía práctica para la elección del presente paquete que más se adecúe a nuestras necesidades y al ordenador personal de que dispongamos.

27 EL LENGUAJE C, PROXIMO A LA MAQUINA
Lenguaje de programación que se está imponiendo en los microordenadores más grandes, tanto por su facilidad de aprendizaje y uso, como por su enorme potencia y su adecuación a la programación estructurada. Vinculado íntimamente al sistema operativo UNIX es uno de los lenguajes de más futuro entre los que utilizan los micros personales.

28 EL ORDENADOR COMO INSTRUMENTO MUSICAL Y DE COMPOSICION

Análisis de cómo se puede utilizar el ordenador para la composición o interpretación de música. Libro eminentemente práctico, con numerosos ejemplos (que usted podrá practicar en su ordenador casero) y lleno de sugerencias para disfrutar haciendo de su ordenador un verdadero instrumento musical.

29 LA CREATIVIDAD EN EL ORDENADOR. EXPERIENCIAS EN LOGO

El LOGO es un lenguaje enormemente capacitado para la creación principalmente gráfica y en especial para los niños. En este sentido se han desarrollado numerosas experiencias. En el libro se analizan estas experiencias y las posibilidades del LOGO en este sentido, así como su aplicación a su ordenador casero para que usted mismo (o con sus hijos) pueda repetirlas.

30 SISTEMAS OPERATIVOS: EL SISTEMA NERVIOSO DEL ORDENADOR

Características de diversos sistemas operativos utilizados en los ordenadores personales y caseros. Se trata de llegar al conocimiento, ameno, aunque riguroso, de la misión del sistema operativo de su ordenador, para que usted consiga sacar mayor rendimiento a su equipo.

NOTA: Ediciones Siglo Cultural, S. A., se reserva el derecho de modificar, sin previo aviso, el orden, título o contenido de cualquier volumen de la colección.

Uno de los lenguajes de programación más en auge de la actualidad es, sin duda, el PASCAL. Este libro hace una introducción a la programación con el lenguaje PASCAL pensada tanto para programadores sin experiencia como para aquéllos que, habiendo probado otros lenguajes, desean aprenden a escribir programas de ordenador de una manera más clara y sencilla.

A lo largo del libro se van introduciendo, además, conceptos básicos de la Informática, como son la ordenación de tablas, los algoritmos recursivos y la gestión de listas encadenadas y árboles.